EDICIONS VIRTUALS



Lluís Gil Espert
Montserrat Sánchez Romero

El C++ por la práctica

Introducción al lenguaje y su filosofía

EDICIONS UPC

Lluís Gil Espert
Montserrat Sánchez Romero

El C++ por la pràctica

Introducción al lenguaje y su filosofía

Primera edición: septiembre de 1999

Edicions de la Universitat Politècnica de Catalunya, SL
Jordi Girona Salgado 31, 08034 Barcelona
Tel.: 934 016 883 Fax: 934 015 885
Edicions Virtuals: www.edicionsupc.es
e-mail: edupc@sg.upc.es

Producción: CBS – Impressió digital
Pintor Fortuny 151, 08224 Terrassa (Barcelona)

Depósito legal: B-33.738-99
ISBN: 84-8301-338-X

Prólogo

Los autores iniciaron hace un par de años en el marco de la Universidad Politécnica de Cataluña unos seminarios para la programación orientada a objeto con lenguaje C++ y aplicada al método de los elementos finitos y el cálculo matricial. De la experiencia, se observó que los asistentes, mayoritariamente ingenieros, tenían una buena base de conocimientos teóricos, pero un dominio débil de los lenguajes de programación, en particular del C++. De hecho, sólo unos pocos sabían algo de C y de programación estructurada. El porqué de esta situación cabe buscarlo en la formación de la ingeniería, que no ahonda en los lenguajes de programación, dejando este campo libre a la autoformación o la especialización posterior de postgrado. Este libro nace, pues, de una necesidad docente.

El objetivo del libro es familiarizar al lector en la programación con lenguaje C++. En general, cuando uno estudia C++, tiene la impresión de que los manuales precedentes están orientados a personas que dominan conceptos avanzados de programación como son la aritmética de punteros, la abstracción de datos, etc., en definitiva, que tienen, por lo menos, algún conocimiento previo de C. Creemos que esto es así por dos motivos; en primer lugar porque el propio C++ es una extensión de C (en el libro de Stroupstrup se lee "*Los buenos programas en C tienden a ser programas en C++*"), y en segundo lugar, por el deseo que tiene todo autor de transmitir en el mínimo de tiempo y espacio toda esa potencia de desarrollo que conlleva el lenguaje. Esto hace que los libros de C++ sean complicados para el lector con poca experiencia en el campo de la programación.

En el libro que el lector tiene entre manos **no se parte de ningún conocimiento preestablecido**, se considera que nadie sabe nada, ni de programación ni de lenguajes, y si sabe algo, mejor para él. Esto significa que el orden de los contenidos, a veces, se aleja un tanto de la bibliografía común. En particular, el lector experimentado pensará que se da un rodeo para ir a un cierto punto en lugar de tirar por la vía directa. Tal vez sea así, en ocasiones, pero nuestra experiencia docente nos permite asegurar que éste es el buen camino. Cuando un niño aprende un lenguaje, empieza por sonidos y palabras sencillas que, a veces, no tienen ni siquiera sentido en sí mismos, pero le sirven para tender puentes hacia la expresión oral y escrita del día de mañana. La similitud con el lenguaje, de ahí su nombre, de programación es evidente; toda obra maestra nace de cientos de cuartillas malbaratadas.

Otro elemento diferencial del libro se encuentra en su estructura. El libro no contiene un texto clásico, sino que **se organiza en fichas de trabajo,** para aprender, como indica el título, de forma práctica. Pensamos que la mejor manera de dominar un idioma extranjero es hablando con los nativos; por consiguiente, para escribir en un lenguaje de programación se debe hablar con la máquina. Ese diálogo hombre-máquina sólo puede hacerse con el ordenador delante y probando la escritura de pequeños programas a modod de ejemplos y ejercicios que puedan controlarse.

Como último punto, cabe subrayar que el libro está dirigido a **todas las personas** que deseen aprender C++, independientemente de su formación básica. Esperamos que el lector disfrute aprendiendo, enfadándose con la máquina y consigo mismo, que sufra y que goce, y que, en definitiva, utilice el libro como una guía para crecer por sí mismo.

Finalmente, agradecer a los ingenieros Klaus Reimann y Orlán Cáceres el interés en la lectura, las sugerencias y las correcciones sobre el manuscrito original. En especial, por su amistad, dedicación y paciencia.

Barcelona y Terrassa 1998

Índice

Capítulos **páginas**

Presentación

En el mundo de la docencia es muy común el método de trabajo con fichas, sobre todo en la enseñanza de idiomas extranjeros. En este caso, aprovechando la similitud que existe entre los idiomas que se utilizan para comunicarse entre personas y los lenguajes de programación que se utilizan para comunicarse con los ordenadores, se ha pensado que dicha metodología docente puede aprovecharse, de forma conveniente, para aprender el lenguaje de programación C++ de manera práctica, rápida e intuitiva.

La única forma de aprender a programar es programando, por ello las fichas proponen ejemplos y ejercicios que dan un enfoque muy práctico a la programación en C++ y se acompañan de los conceptos teóricos necesarios para dominar la sintaxis y la filosofía del C++.

A continuación se describe el contenido de las fichas y cómo se deben trabajar los diferentes apartados. Asimismo, al final de este capítulo, se explica brevemente cómo se crea un programa de ordenador; para que el lector sea capaz de reproducir los códigos que se acompañan.

Antes de empezar a trabajar con las fichas será necesario que usted consiga un ordenador y un compilador de C++. Existen diversos productos comerciales, desde Visual C++ hasta Borland C++, y también compiladores *freeware*, por ejemplo los de gnu. En cualquier caso, su opción vendrá limitada por el sistema operativo de la máquina y las necesidades que usted se imponga.

1 Descripción de las fichas

Toda ficha se inicia con un **título** y en general, salvo alguna introducción previa, se definen inmediatamente los **objetivos** docentes que se pretenden conseguir. Posteriormente, se suministra un **código de trabajo** donde aparecen los **conceptos** que se quieren ilustrar. Éstos se desarrollan extensamente en las líneas siguientes, estando acompañados de comentarios que hacen referencia al código de trabajo.

A continuación, se proponen **ejercicios** y un **ejemplo**, sobre de matrices numéricas, que se irá desarrollando y volviendo más complejo a medida que se adquieran más conocimientos. Finalmente, y sólo en algunos capítulos, se incluye una **ampliación de conceptos**.

Más o menos en cada ficha se encuentran los apartados que vienen a continuación.

Ejemplo de ficha

Objetivos generales

En este apartado se citan de forma breve y clara los objetivos de la ficha, es decir, aquello que se espera que usted sepa o pueda hacer al terminar al ficha.

Código de trabajo

Para llegar a los objetivos, se presentan en cada ficha uno o más códigos básicos de trabajo. El código de ordenador se encuentra dividido en dos columnas; en la primera se numeran las líneas de código para facilitar los comentarios posteriores y en la segunda se escribe el código con las instrucciones que se suministra a la máquina. Algunas palabras están en negrita porque se relacionan con los conceptos que se debe trabajar.

Por ejemplo :

Tal y como aparece en la obra

```
...
211 // es un ejemplo
212 return 10 ;
...
```

Líneas	**Código real**
...	...
211	// es un ejemplo
212	return 10 ;
...	...

¡Atención! Porque, para crear un programa, no se deben escribir los números de línea.

Conceptos

Los conceptos teóricos y prácticos que se encuentran en el código, y que son motivo de estudio para cumplir los objetivos de la ficha, se presentan con explicaciones y aclaraciones detalladas que le introducirán en el lenguaje de forma progresiva.

Por ejemplo :

1 La instrucción return

Cuando una función termina, suele devolver un cierto valor a través del la palabra clave *return*, tal y como se muestra en la línea **212,** etc.

Ejercicios

En este apartado se proponen ejercicios que usted debe desarrollar de forma individual. La generación de código para resolver los problemas propuestos le servirá para formularse nuevas preguntas y consolidar los conceptos adquiridos. Todos los ejercicios están resueltos porque se aprende tanto hablando como escuchando, y el observar códigos ya existentes es una forma muy buena de aprender; sin embargo, se recomienda que antes de ver la solución intente encontrarla por usted mismo.

Ejemplo

A lo largo de toda la obra se trabajará un ejemplo sobre matrices y vectores, se verá evolucionar la programación sobre dichas entidades matemáticas a medida que se tengan más conocimientos.

Ampliación de conceptos

En algunas fichas, se explicarán detalles que no aparecen explícitos en los apartados anteriores y que se consideran interesantes para complementar y ampliar el conocimiento del lenguaje. Este apartado es mejor dejarlo para revisiones posteriores de las fichas, su lectura puede complicarle un poco el proceso de aprendizaje; no obstante está ahí para que sepa las posibilidades que existen y las pueda utilizar en su momento.

2 Los programas de ordenador

Es posible que usted nunca haya oído hablar de conceptos como compilación, ejecutables, etc. No se trata de detallar aquí los mecanismos por los cuales un código escrito en un lenguaje de programación se convierte en un programa que hace ciertas cosas en un ordenador. El tema es en sí mismo motivo de libros. Sin embargo, sí que es interesante explicar cómo se obtiene un programa a partir de un código.

Para obtener un programa que se pueda ejecutar en un ordenador se necesita un código fuente, es decir, un archivo de texto con las instrucciones. Este archivo suele tener la extensión .cpp y es el que usted tiene que escribir como programador. También son necesarios unos archivos de cabecera con la extensión .h; de éstos, algunos serán escritos por el programador, pero otros ya vienen con el compilador. Con todos los archivos se realiza el proceso de compilación que da como resultado un archivo de extensión .obj.

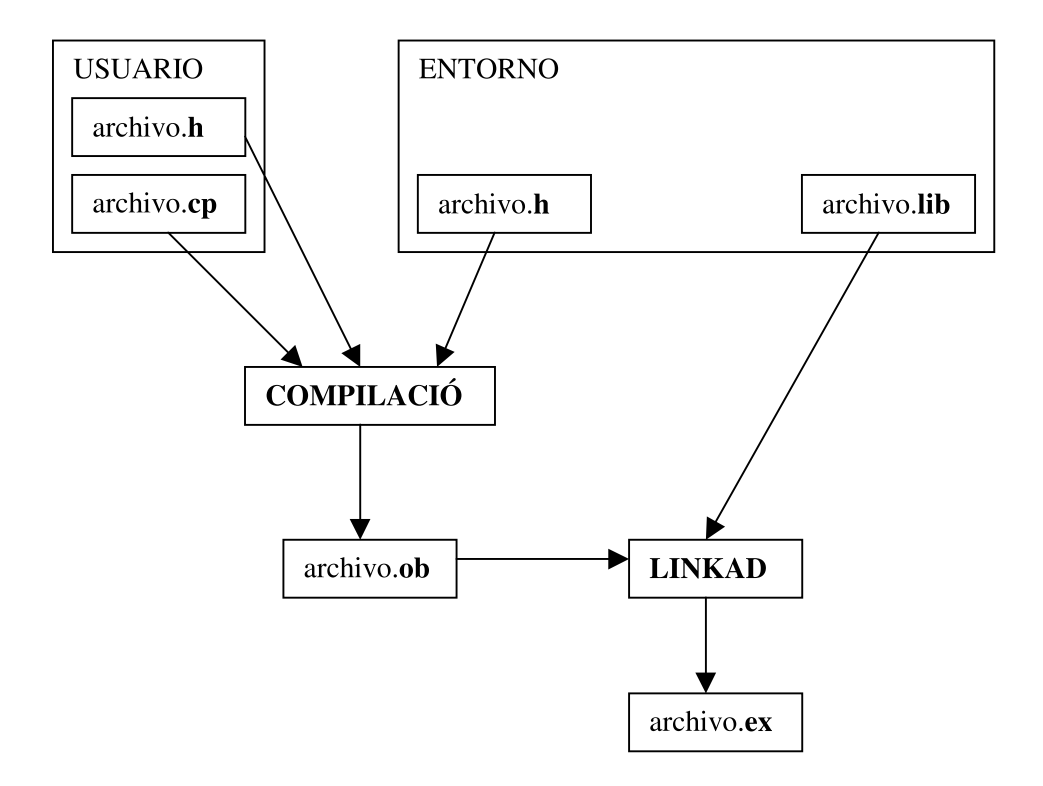
El código que el programador escribe lo entienden las personas, pero no la máquina. El ordenador tiene su propio lenguaje formado por unos y ceros, que es complicado para las personas. Entonces, qué se entiende por compilación. Pues simplemente la traducción de todas las instrucciones al idioma de la máquina. El programador se libera del complicado lenguaje de la máquina y se concentra en el lenguaje de programación mucho más cercano a la forma de pensar de los humanos.

Finalmente, antes de obtener el ejecutable es necesario un linkaje que enlaza el archivo .obj con las librerías que suministra el lenguaje. La mezcla del archivo .obj con las librerías .lib conducen a un ejecutable .exe. Este último archivo es el que se podrá ejecutar en la máquina.

Hoy en día, hay entornos de programación que realizan está labor en un conjunto integrado, posiblemente usted trabaje con uno de ellos. En tal caso le remitimos al manual de instrucciones para compilar y ejecutar un archivo. En general, de todas las funcionalidades que tienen esos entornos usted sólo va a necesitar unas pocas; por lo tanto le aconsejamos que no pierda mucho el tiempo en aprender su manejo hasta que realmente no necesite hacer cosas muy potentes.

Trabaje con un único archivo .cpp y a lo sumo un archivo .h, el resto lo pone todo el entorno. Construya y ejecute las aplicaciones, no necesita nada más. Bueno, un poco de suerte.

Fig. 1: Esquema del proceso de compilación y linkado para obtener un ejecutable.



Después de esta breve presentación de las fichas de trabajo, buena suerte y adelante.

Ficha 1 : Sintaxis básica

1 Objetivos generales

- Conocer cuál es la estructura básica del lenguaje para poder escribir un programa.
- Iniciarse en el manejo del entorno de compilación.

2 Código de trabajo

```
//Ficha 1
/* un programa que pregunta su nombre y le saluda */

#include <iostream.h>
#define FRIEND 1

int main (void)
{
char name [200] ;
cout <<"escribe tu nombre"<<endl ;
cin >> name ;

#ifdef  FRIEND
cout << "Hola "<< name << " , que tal !"<<endl ;
#endif

 return 0 ;
}
```

3 Conceptos

3.1 Esqueleto básico : función *main*, las llaves { } y el ;

Todo programa es un conjunto de instrucciones que la máquina debe ejecutar en un determinado orden. De forma general y sin entrar en detalles, se puede afirmar que las instrucciones básicas se inscriben en la función *main.* Por lo tanto, una vez se ha iniciado el programa, el ordenador busca la función *main* y ejecuta todas las instrucciones que encuentra hasta que dicha función termina. Esta visión, un tanto simplista, pretende concienciar sobre la naturaleza fundamental de la función *main***.** Simplemente, todo programa escrito en C++ debe contener una función *main***.**

En el código de ejemplo, la máquina inicia las instrucciones a partir de la función *main,* en este caso la línea **005**. Por el momento, para no complicar la exposición se ignorará el sentido de las palabras *void* e *int* y se deja dicha explicación para fichas posteriores.

Una vez se ha detectado la función *main***,** el ordenador ejecutará de forma secuencial todo lo que encuentre entre las llaves {} que marcan consecutivamente el inicio y el final del programa. En consecuencia, el programa se inicia en la línea **006** y termina en la línea **014**. Las {} se utilizan para formar bloques de código; en este caso el propio programa es un bloque, pero en casos más complejos puede existir más de un bloque.

Nótese que casi todas las líneas terminan con el símbolo *;* de punto y coma. Esto indica que termina una instrucción para la máquina; por ejemplo, la línea **009** está leyendo el nombre del usuario. El concepto sería algo así como terminar una frase con punto y coma para que la otra persona sepa que ese mensaje ha terminado y va a empezar otro. La omisión del *;* conduce a errores en la compilación del programa.

En resumen, la función *main* es necesaria en todo programa en C++, las llaves {} se utilizan para marcar bloques de código y el *;* señala líneas de código.

3.2 Realización de comentarios

Las líneas **001** y **002** son comentarios, es decir, líneas que el compilador no traduce en instrucciones para la máquina, sino que es texto escrito que no se tiene en consideración. Sin embargo, el hecho de que el compilador no las tenga en cuenta no quiere decir que no sean importantes, muy al contrario, son vitales.

Imagine que usted crea un programa que funciona correctamente y que deja pasar un cierto tiempo hasta volver a tocarlo. Si el programa es pequeño posiblemente no tendrá muchos problemas para recordar qué hacía, cuáles eran las variables, etc. No obstante, a medida que su programa crezca, cada vez le resultará más difícil saber qué hacía cada parte del programa, y si el programa es fruto de un equipo de trabajo... Seguro que es tarea de locos intentar modificarlo si no encuentra comentarios que le ayuden.

Por lo tanto, piense que el código lo utiliza el compilador para que la máquina haga cosas, pero los comentarios los utilizan los programadores para entender el código con facilidad. Unos buenos comentarios hacen comprensible el programa y además pueden utilizarse para elaborar manuales de usuario con sólo un poco más de esfuerzo.

3.3 Archivos de cabezera. Las funciones : *cin* y *cout*

En la línea **003** se informa al compilador que el archivo de cabecera es *iostream.h.* Este archivo contiene información acerca de las funciones estándar de C++ que se utilizarán en el programa, en particular las funciones *cin* y *cout* que permiten escribir y leer datos. Todo compilador se acompaña de otros archivos de cabecera como string.h, math.h, etc. que definen funciones que ayudan a trabajar con funciones de cadenas, matemáticas, etc. respectivamente.

Las funciones *cin* y *cout* se utilizan para que el usuario interaccione con el programa; la descripción detallada de la funciones se deja para más adelante y en este caso *cin* lee un nombre y *cout* escribe en pantalla.

4 Ejercicios

4.1 Escribir el mensaje en pantalla "esto del c++ parece muy fácil"

La solución más sencilla sería :

```
#include <iostream.h>

int main (void)
{
cout << "esto del C++ parece muy fácil" << endl ;
return 0 ;
}
```

Una opción un poco más sofisticada :

```
#include <iostream.h>

int main (void)
{
char msg[200] ;
cout<<"escribe un mensaje con una sola palabra:"<<endl ;
cin >> msg ;
cout<<"********************"<<endl ;
cout<< msg << endl ;
return 0 ;
}
```

5 Ejemplo

En este caso sólo se puede escribir algo así :

```
#include <iostream.h>

int main (void)
{
cout << "Vamos a trabajar con matrices y vectores"<<endl ;
return 0 ;
}
```

6 Ampliación de conceptos

6.1 Predirectivas de compilación

A veces es interesante que en tiempo de compilación parte de su código se compile y otras partes queden ocultas, por ejemplo si se desea regalar unas versiones mutiladas como demostración, o bien si se debe compilar en entornos diferentes. También es interesante fijar valores de constantes que no van a cambiar a lo largo del programa, como puede ser el número PI.

Para ello se dispone de las predirectivas de compilación, como la línea **004,** donde se asigna un valor a una variable llamada *FRIEND.* Para entender cómo funciona, le sugerimos que anule dicha línea con un comentario y recompile para observar cómo se comporta ahora el programa.

Las directivas se interpretan en la primera pasada del compilador y desaparecen en las siguientes. Son información que le pasamos al compilador para que realice alguna operación. LLevan un # delante y no van seguidas de punto y coma.

Las más habituales son:
define que se utiliza para definir alguna variable como en la línea **004**.
undef que desactiva la definición previa.

#ifdef que realiza una evaluación lógica, si está definida la variable entonces vamos a continuar; el bloque lógico se cierra con *#endif*; esto aparece en las líneas **010** y **012**.

El conjunto de sentencias lógicas más comunes son:
#if si condicional, *#ifdef* si está definido, *#ifndef* si no está definido, *#elif* en caso contrario si, *#else* en caso contrario, *#endif* final del bloque lógico.

Es muy recomendable escribir sus archivos de cabecera con el siguiente formato para evitar errores en tiempo de compilación:

```
#ifndef _MY_NOMBRE_DE_FICHERO_H
        // donde _MY_NOMBRE_DE_FICHERO_H es el nombre que quiera dar a una variable
#define _MY_NOMBRE_DE_FICHERO_H

{ escriba aquí todo su código }

#endif   //_MY_NOMBRE_DE_FICHERO_H
```

Ficha 2 : Tipos básicos

1 Objetivos generales

- Conocer qué tipos de datos básicos existen por defecto.
- Conocer las operaciones que permiten estos datos.

2 Código de trabajo

```
001    //Ficha2
002    /* se presentan algunos tipos de datos básicos */

003    # include <iostream.h>

004    int global = 5;

005    int main (void)
006    {
007    int a =1 ;
008    double x = 1.2 ;
009    double y = 2.3 ;

010    cout << "x+y="<<x+y<<" "<<"a="<<a<<endl;
011    a=global + (int) x;
012    cout<<"a="<<a<<endl;
013    return 0;
014    }
```

3 Conceptos

3. 1 La importancia de los datos

Los programas están compuestos por entidades de dos naturalezas, los datos y manipulaciones sobre los datos, llamadas procedimientos. En general, los procedimientos conducen a resultados a partir de los valores de los datos. Esto es un programa: unos valores iniciales de los datos de entrada y unos valores finales de los datos de salida. Por ejemplo, en las líneas **008** y **009** se definen dos variables (entrada) y en la línea **010** se calcula (operación) y se escribe el resultado de su suma (salida).

Dado que cualquier problema se puede expresar como un conjunto de variables o parámetros y unas secuencias de órdenes que ejecutar, la tarea del programador es decidir cuáles son los datos fundamentales y cómo se organiza la información; posteriormente se debe definir que operaciones se

realizarán sobre los datos para obtener un resultado. Ésta es la gran diferencia entre el enfoque moderno de la programación y los planteamiento clásicos, los datos son más importantes que los procedimientos.

Distinguimos dos tipos de datos: *tipos fundamentales* y *tipos derivados*. Los datos de *tipo fundamental* son los datos básicos que por defecto tiene el lenguaje y que reconoce el compilador.

Los *tipos derivados* son aquellos datos que el programador construye a medida a partir de los fundamentales o de otros derivados. Éste es el caso de las estructuras, uniones, clases, arrays, etc. cuyo estudio detallado se irá viendo en fichas posteriores.

3.2 Declaración e inicialización de variables

Una variable se declara dándole un nombre y asignándole un tipo; por ejemplo, en la línea **008** se define una variable de nombre *x* y del tipo *double*. Por lo tanto, se está indicando qué valores puede tomar (real de doble precisión) y qué operaciones se pueden realizar con ella, suma y escritura en la línea **010**.

Un concepto muy importante y sobre el que se insistirá en la ficha 7, es la definición de *ámbito* o *alcance (scope)* de una variable. Según dónde se declare una variable, ésta puede tener un ámbito *local* o *global*. Se entiende por variable *global* aquella que es accesible y conocida desde cualquier punto del código. En cambio, variable *local* sería aquella que sólo puede ser accesible en una parte del código. En general y de forma simplificada, se asigna el ámbito de una variable al trozo de código comprendido entre su definición y la próxima } que cierre un bloque de código.

En particular, en el código de trabajo se puede observar que en la línea **004** se ha declarado una variable antes de la función *main*; en consecuencia, será una variable *global* accesible desde la línea **011**. Por el contrario en las líneas **007, 008** y **009** se declaran variables *locales*, la primera de tipo *int* y las dos siguientes de tipo *double*, que sobreviven hasta la línea **014**.

Se define inicialización como la acción de asignar un valor inicial a una cierta variable, línea **004** por ejemplo. Lógicamente, este valor puede ir cambiando a lo largo de la ejecución del programa. Algunos compiladores inicializan automáticamente las variables *globales* a cero pero no las variables *locales*, por ello es *recomendable dar siempre un valor inicial a todas las variables de un programa*. No seguir esta regla suele conducir a errores en tiempo de ejecución.

3.3 Operaciones y procedimientos asociadas a los tipos de datos

Un *operador* es un símbolo que indica cómo deben ser manipulados los datos. Por ejemplo, los simbolos +,-,*,etc. Todos los datos tienen asociados unos operadores de forma natural. Las operaciones asociadas a los datos de tipo fundamentales son conocidas por el compilador y no es necesaria una definición previa de ellas, por ejemplo la suma de enteros, resta de reales, etc. Posteriormente, se verá que cuando se trabaja con tipos de datos *definidos por el usuario* será necesaria una redefinición de las operaciones asociadas a ellos.

En la línea de código **010** se realiza la operación de suma de dos variables *double* y la escritura por pantalla de su resultado. En la línea **011** se suma una variable *int* con la parte entera de una variable *double*, almacenando el resultado en una variable *int* y escribiendo por pantalla el resultado en la línea **012**.

Los *procedimientos* son manipulaciones complejas de los datos y están asociados al concepto de funciones que se explica en la ficha 6. Los procedimientos conllevan operaciones más complicadas que las básicas operaciones aritméticas o lógicas definidas por defecto en el compilador.

4 Ejercicios

4.1 El programa pregunta dos números y muestra el resultado.

Solución propuesta para al ejercicio 4.1.

```
// Ficha 2
/* el programa pregunta dos números y muestra el resultado*/

#include <iostream.h>
int main (void)
{
double x, y, z;
cout << "Introduzca el primer número" << endl ;
cin >> x;
cout << "Introduzca el segundo número" << endl ;
cin >> y;
z=x+y;
cout << x <<"+"<< y <<"="<< z <<endl;
return 0 ;}
```

4.2 Realizar operaciones con tipos diferentes

Solución propuesta para al ejercicio 4.2.

```
//Ficha 2
/* operaciones con tipos diferentes*/

#include <iostream.h>

int main (void)
{
int i=1;
double x=-1.26;
double y=2;

x+=y;    //x=x+y, suma double de dos doubles, x=0.74
y*=y;    //Y=Y*Y, producto double de dos doubles, y=4
x=x+i;   //suma double de un double y un entero, x=1.74
i+=x;    // suma entera de un entero y un double, i=2

cout<<"x+y"<<x+y<<"i="<<i<<endl;  // x+y=5.74  i=2

return 0 ;}
```

4.3 ¿Se pueden sumar dos carácteres? ¿Qué se obtiene?

Solución propuesta para al ejercicio 4.3.

```
//Ficha 2
/* ¿Se pueden sumar dos caracteres?*/

#include <iostream.h>

int main (void)
{
char a; //declaracion de variable tipo caracter
char b;

cout<<”Introduzca el primer caracter:”<<endl;
cin >> a;

cout<<”Introduzca el segundo caracter”<<endl;
cin >> b;

a+=b;// a=a+b, suma tipo caracter de dos caracteres
cout <<“el resultado “<<a;
return 0 ;}
```

5 Ejemplo

Se define la dimensión de un vector mediante un entero, el concepto matriz se restringe de momento a un vector:

```
#include <iostream.h>

int main (void)
{
 int size=3;
 cout << "Vamos a trabajar con un vector de " << size
       << " componentes" <<endl;
 return 0;
}
```

6 Ampliación de conceptos

6.1 Tipos fundamentales

Los datos de *tipo fundamental* se pueden clasificar en:

Tipos enteros: *char* (carácter) *short* (formato corto), *int* (entero), *long (*formato largo) y *enum* (enumeración).
Tipos reales: *float* (precisión simple), *double* (precisión doble) y *long double* (formato largo).
Otros : *void* (vacío).

Los tipos enteros y reales se suelen llamar aritméticos. Los valores mínimo y máximo de cada uno de los tipos están incluidos en el fichero <limits.h> y son dependientes de cada implementación y procesador. Los tipos reales se usan para operaciones matemáticas y científicas. Cada implementación del lenguaje definirá las características de los tipos reales en el fichero <float.h>.

La codificación de un entero suele depender del tamaño de palabra del procesador donde se va a ejecutar el programa generado por el compilador, variando entre 16 y 64 bits. Para poder tener distintos tamaños de enteros se pueden usar los tipos modificador *short* y *long*, que se puede usar acompañando a *int*, o a solas, asumiendo que se está hablando de enteros. El modificador *short* suele reducir el tamaño de palabra a 16 bits y el modificador *long* lo suele aumentar a 32.

Cada uno de los tipos enteros puede ser utilizado con las palabras clave *signed* y *unsigned*, que indican respectivamente si es un entero con signo o sin él, en este último caso se considera como un valor entero positivo. Por defecto se asume *signed*. Esto significa que, por ejemplo, para 16 bits, los valores fluctúan entre 32768 y -32767, ambos inclusive. En cambio, para tipos *unsigned*, fluctúa entre 0 y 65535.

El tamaño en bits de los tipos reales es muy variable, dependiendo mucho del ordenador, procesador, coprocesador matemático presente, etc. Lo único que se debe tener en cuenta es que la resolución de *double* es mayor o igual que la de *float* siendo dos tipos compatibles. También existe un tipo *long double* con una precisión mayor.

6.2 Caracteres

La codificación de un carácter depende del ordenador y suele variar entre 7 y 9 bits, siendo el valor más habitual el uso de caracteres de 8 bits (1 byte). La codificación suele seguir la norma ASCII (*American Standard Code for Interchange of Information)* y por tanto tendremos 256 caracteres.

Una constante de tipo *char* se escribe entre apóstrofes, como ´x´. Las constantes de tipo *char* se pueden sumar, restar, multiplicar , etc, ya que son de tipo aritmético. El valor ASCII u otro código de cada carácter no es necesario, ya que como hemos dicho, los caracteres se pueden operar como si de números se tratara.

6.3 Conversión de tipos

Cuando se opera con datos de diferente naturaleza, ambos son convertidos al tipo de dato de precisión más alta y el resultado es convertido al tipo de variable que lo almacena. Por ejemplo, si se suma una variable *a* entera con una variable *b* double y se almacena el resultado en una variable entera *c,* entonces sucederá lo siguiente: primero se convertirá la variable *a* en double, se realizará la suma entre doubles y finalmente se almacenará la parte entera del resultado en la variable *c*. Si *c* hubiese sido una variable tipo carácter, se convierte a su código ASCII..

En general, no es conveniente dejar en manos del compilador la conversión de tipos. El programador puede forzar la conversión de tipos mediante un *casting,* para ello basta con escribir el tipo entre paréntesis delante de la variable. En la línea de código **011** se realiza una conversión temporal de la variable de tipo double *x* en una de tipo *int* mediante el *cast (int).*

6.4 Operadores

A continuación resumimos en una tabla los operadores del C++. Están agrupados por orden de precedencia, donde cada caja tiene mayor precedencia que las cajas de abajo.

Algunos operandos ya se estudiarán en capítulos posteriores, sobre todo los relacionados con las clases.

Operadores	Descripción	Sintaxis
++	postincremento	Ivalue + +
++	preincremento	+ + Ivalue
- -	postdecremento	Ivalue - -
- -	predecremento	- - Ivalue
*	multiplicación	expr * expr
/	división	espr / expr
%	resto	expr % expr
+	suma	expr + expr
-	resta	expr - expr
<	menor que	expr < expr
< =	menor o igual que	expr < = expr
>	mayor que	expr > expr
> =	mayor o igual que	expr > = expr
= =	igual que	expr = = expr

Ficha 3 : Sistemas E/S y lectura de ficheros

1 Objetivos generales

- Ser capaces de interaccionar con el programa escribiendo y leyendo datos desde consola y desde ficheros.

2 Código de trabajo

```
//Ficha 3
#include <iostream.h> // librería para manejar la E/S de tipos
                      // predefinidos

int main (void)
{
int outval;
cout << "escribe un valor = " ;  //outval = 4325 por ejemplo
cin >> outval;

cout << "el valor escrito es = " << outval << endl;

return 1;
}
```

3 Conceptos

3.1 Introducción sistemas E/S (entrada/salida)

Se considera fundamental que el programador pueda interaccionar mínimamente con el ordenador, antes de continuar adelante. Por ello, se presentan los fundamentos de lectura y escritura como soporte para el aprendizaje. Sin embargo, hay que advertir que la librería de C++ de lectura y escritura no es nada trivial y el lector deberá utilizar símbolos y operadores que a estas alturas no tienen ningún sentido para él. Se ruega que se acepten como receta de cocina y serán explicados con detalle en próximas fichas.

La totalidad de lenguajes disponen de una librería de funciones para entrada/salida (E/S) que permiten la comunicación entre el usuario y el programa. Las opciones de entrada y salida suelen estar limitadas a los tipos de variable definidos por defecto en el lenguaje; por ejemplo, la librería de E/S del C es la **<***stdio.h***>** y comprende funciones para la lectura y escritura de enteros, reales, cadenas, etc.

El C++ es un lenguaje expandible, en el sentido de que el usuario puede definir sus propios tipos de variables; se hace por tanto necesario una librería que se comporte igual para todos los tipos. Por lo tanto, debe ser una librería flexible y eficiente, adaptable a las necesidades del programador. La librería incluida en el C++ para manejar la E/S es la *iostream.* Esta librería se puede incorporar en cualquier programa incluyendo el archivo de cabecera *iostream.h* tal y como muestra la línea **001**.

El concepto que se utiliza en C++ para manejar la E/S es el concepto de *stream* (flujo). Los datos de lectura y escritura se considera que crean una corriente que actúa como fuente (entrada) y/o destino (salida). La librería de *streams* consta de unos veinte tipos predefinidos que permiten tanto la E/S por terminal como por ficheros y *strings* (cadenas).

3.2 La consola

El instrumento más rudimentario de interacción es la consola, es decir, el teclado para permitir la interacción hombre-máquina y la pantalla para la relación inversa. En la librería <*iostream.h*> hay definidos cuatro elementos de interacción.

cout	Salida por pantalla. En el código en la línea **005**, se define un stream de salida en pantalla imprimiendo el mensaje: *escribe un valor* =.
cin	Entrada por el teclado. En la línea **006**, el usuario puede introducir un valor desde el teclado, el dato se almacena en la variable *outval*.
cerr	Salida de error por pantalla.

Los *operadores* **<<** y **>>**, son los llamados *operadores de inserción y de extracción* respectivamente, sirven para dirigir el flujo de datos. En la línea **005**, el mensaje se dirige del programa a la pantalla (*cout*) y en la **006** del teclado (*cin*) al programa.

La salida se realiza a través del operador << y consiste en insertar los datos en el *stream*, en particular se convierte en salida por la pantalla; por ejemplo, la línea **007** introduce un texto y un número. Lo mismo ocurre con la entrada, el operador >> saca del *stream* el tipo de dato necesario.

4 Código de trabajo

```
//Ficha 3

#include <fstream.h>       //librería para manejar la
                          //lectura/escritura de ficheros
int main (void)
{
int outval=3;
ofstream WriteFile("myfile.txt");      //apertura fichero en modo escritura
WriteFile << outval;           //escribimos el valor de outval en el fichero
WriteFile.close();                                        //cierre de fichero
int inval;
ifstream ReadFile("myfile.txt");     //declaración de un fichero de lectura
ReadFile >> inval;              //leemos el fichero y lo colocamos en inval

ReadFile.close();                                         //cierre del fichero
cout << "el valor escrito es = " << inval << endl;
            //el valor escrito es = 3

return 1;
}
```

5 Conceptos

5.1 Ficheros. Abrir, cerrar, escribir y leer

Igual que la consola se asocia a unos *streams* llamados *cin* y *cout*, se pueden asociar funciones de lectura y escritura a un *stream* para ficheros. El fichero o archivo es un espacio de disco donde se almacena información de forma secuencial bajo un nombre. Para trabajar con ficheros es necesario el archivo de cabecera que aparece en la línea **101** **<***fstream.h***>** el cual incluye a <*iostream.h*>.

Debe destacarse que en el disco duro aparecerá el nombre del fichero, pero el programa utiliza el archivo a través del *stream* que le hayamos asociado. Esto se observa claramente en la línea **105** donde el fichero se llama *myfile.txt*, mientras que el *stream* se maneja bajo la variable *WriteFile*, tal y como muestra la línea **106**.

Las operaciones que se pueden hacer sobre un fichero son: creación, apertura, lectura, escritura y desplazamiento a lo largo de la información que contiene. El uso de ficheros es muy importante para facilitar tanto la lectura de datos como la escritura de resultados. Sin embargo, el acceso a los datos de grandes ficheros puede producir una notable disminución en la velocidad de ejecución del programa, por ello se recomienda usar los ficheros con inteligencia. Las posibilidades que abre el uso de ficheros son muy grandes y escapan a la intención de esta ficha inicial, por ello se van a restringir las acciones sobre un fichero. Las operaciones básicas sobre un fichero son:

Abrir un fichero
En la línea **105** se abre un fichero de nombre *myfile.txt.* El fichero se abre con un tipo *ofstream* (*output file stream*); por lo tanto, la intención es escribir datos en él. Por el contrario, en la línea **109** se define un tipo *ifstream* (*input file stream*); por lo tanto, sólo se desea leer datos de él; en este último caso, el archivo se abre en esa línea bajo el nombre de *myfile.txt* que es el fichero que ya se había creado antes.

Cerrar un fichero
Si un fichero no se usa, es importante cerrarlo porque son recursos que ocupan al sistema y puede producir errores en tiempo de ejecución, si el programa pretende volver a abrirlo. Para cerrar un fichero basta con ejecutar un *close* tal y como muestran las líneas **107** y **111**.

Escritura y lectura de un fichero
Para leer y escribir basta con utilizar los operadores >> y << que ya se presentaron en el apartado de la consola. Por ejemplo, la línea **106** escribe una variable en el fichero y la **110** lo lee y almacena en una variable tipo *int* llamada *inval*.

6 Ejercicios

6.1 El programa realiza operaciones con números y escribe el resultado en un archivo.

A continuación se presenta una solución propuesta para el ejercicio. En este caso, las operaciones se han restringido a la suma y diferencia de enteros. Por supuesto, esto mismo sería aplicable a otro tipo básicos de datos numéricos como los presentados en la ficha 2. Como ejercicio el lector puede intentar la multiplicación, división, etc. con *double*.

```
//Ficha 3
/* el programa realiza operaciones con números y escribe
   el resultado en un archivo */
#include <iostream.h>
#include <fstream.h>

int main (void)
{
int a, b, c;
cout <<"dame el valor de a =";
cin >> a;                 // a=2
cout <<"dame el valor de b =";
cin >> b;                 // b=3

ofstream fp("nombre.dat");
fp << a+b << " ";    // 5
fp << a-b << endl; // -1
fp.close ( );

int sum, dif;
ifstream  fr("nombre.dat");
fr >> sum >> dif;
fr.close ( ) ;

cout <<"la suma es "<< sum <<endl;
cout <<"la diferencia es "<< dif <<endl;
return 0;
}
```

7 Ejemplo

Se define la dimensión de un vector mediante un entero, el concepto matriz se restringe de momento a un vector y sólo se escribe un texto de salida.

```
#include <iostream.h>
#include <fstream.h>

int main (void)
{
 int size;

 cout << "Introduzca la dimensión del vector"<<endl;
 cin >> size;
 cout << "Vamos a trabajar con un vector de " << size
       << " componentes" <<endl;

 ofstream MyReport("resultado.txt");
 MyReport << " *********************************** inicio" << endl;
 MyReport << " Vector de " << size << " componentes" << endl;
 MyReport << " ************************************** fin" << endl;
 MyReport.close();

 return 0;
}
```

Ficha 4a : Control de flujo. Condicionales I

1 Objetivos generales

- Conocer las sentencias *if/else* condicionales que permiten tomar decisiones.

2 Código de trabajo (if / else)

```
//Ficha 4a
/*Sentencias tipo if/else. Anidamientos y operaciones lógicas*/
#include <iostream.h>
int main (void)
{
int i=1; int j=2;
if (i<j)
{cout <<"i es menor que j"<<endl;}
else
{cout <<"i es mayor que j"<<endl;}
return 0;
}
```

3 Conceptos

3.1 Sentencias tipo if-else

La sentencia *if* permite a un programa tomar una decisión para ejecutar una acción u otra, basándose en el resultado verdadero o falso de una expresión.

La sintaxis de esta sentencia es la siguiente:

```
if (condición)
{
    sentencia 1;        //hacer algo
    sentencia 2;
}
else
{
    sentencia 3;        //hacer otra cosa
    sentencia 4;
}
```

La ejecución de una sentencia *if* consiste en:

1º) evaluar la condición entre paréntesis.
Esta condición puede ser una expresión numérica (ejemplo, la variable x debe tener un valor diferente de cero, lo cual se escribiría como *x != 0*), de relación (el valor de la variable i debe ser menor que el de la variable j, tal como se indica en la línea de código **007**) o lógica (el valor de la variable i y el de j debe ser diferente de cero, lo cual se escribiría como *i!=0 && j!=0*, o bien de forma más simplificada *i && j*).

2º) si el resultado de evaluar la condición es verdadero (resultado distinto de 0), se ejecutan las sentencias 1 y 2 (sentencias encerradas entre llaves después de la cláusula *if*). En el código de trabajo, si se cumple la condición especificada en **007**, se ejecutará la sentencia de la línea **008**, imprimiéndose por pantalla el mensaje "i es menor que j".
.
3º) si el resultado de evaluar la condición es falso (resultado cero), no se ejecutan las sentencias 1 y 2 y se pasa a ejecutar las sentencias especificadas después de la cláusula *else* (sentencias 3 y 4). En el código de trabajo, si no se cumple la condición **007**, se ejecutará la sentencia **010** y se escribirá por pantalla el mensaje "i es mayor que j".

Es tarea del programador estructurar correctamente los *if* y evaluar sus condiciones para que el programa de resultados correctos. Por ejemplo, ¿qué sucede si los números i-j coinciden? ¿Cuál de las dos condiciones se ejecuta?

4 Ejercicios (if / else)

4.1 El programa pregunta dos números, qué operación desea hacer, y muestra el resultado.

Una posible solución al ejercicio

```
//Ficha 4a
/*pregunta dos números, que operación deseas hacer y muestra resultado*/
#include <iostream.h>
int main (void)
{
double a, b;
int c;
cout << "Introduzca el primer número" <<endl ;
cin >> a;
cout << "Introduzca el segundo número" <<endl ;
cin >> b;
cout << "Que operación deseas hacer, 1(suma), 2(producto)" <<endl ;
cin >> c;

if (c==1) {cout <<"el resultado es:"<<a+b<<endl;}
else      {cout <<"el resultado es:"<<a*b<<endl;}
return 0 ;
}
```

4.2 Calcular áreas de diferentes polígonos.

En el siguiente código se presenta una solución para los polígonos más sencillos: triángulo, cuadrado y rectángulo.

```
//Ficha 4a
/* Calcular áreas de diferentes polígonos*/
#include <iostream.h>
int main (void)
{
double a, b;
double Area;
int figura;
cout<<"figura = cuadrado(1), rectángulo(2), triángulo(3)"<<endl;
cout<<"¿que área quieres calcular?"<<endl;
cin>>figura;

if (figura==1)
{ cout <<"valor del lado="; cin >>a; Area=a*a;}

else if (figura==2)
{cout <<"valor del lado menor =";
 cin >> a;
 cout <<"valor del lado mayor =";
 cin >>b;
 Area=a*b;}

else if (figura==3)
{cout <<"valor de la base =";
 cin >>a;
 cout <<"valor de la altura =";
 cin >>b;
 Area=a*b/2.0;}

else
{cout <<"figura equivocada"<<endl;
 Area=0;}

cout<<"El área de la figura es="<<Area<<endl;
return 0 ;}
```

5 Código de trabajo (operadores lógicos)

```
//Ficha 4a
/*Operadores lógicos*/

#include <iostream.h>

int main (void)
{
int a=5; int b=0;
int d = a&&b;  cout << d; // operador && AND
int e=a||b;    cout << e; // operador || OR
int f=!a;      cout << f; // operador ! NOT
return 0;
}
```

6 Conceptos

6. 1 Operadores lógicos

En C++ una expresión es verdadera si devuelve un valor distinto de cero, y falsa en caso contrario. Por lo tanto, el resultado de una operación lógica (AND, OR, NOT) será un valor verdadero (1) o falso (0).

Los operadores lógicos en C++ son los siguientes:

Operador && (AND)
 (expresión && expresión)
* Da como resultado el valor lógico 1 si ambas expresiones son distintas de cero.
* Da como resultado el valor lógico 0 si alguna de las dos expresiones es igual a cero. Si lo es la primera, ya no se evalúa la segunda (línea de código **107**).

Operador || (OR)
 (expresión || expresión)
* Da como resultado el valor lógico 1 si una de las expresiones es distinta de cero. Si lo es la primera, ya no se evalúa la segunda (línea de código **108**).
* Da como resultado el valor lógico 0 si ambas expresiones son cero.

Operador ! (NOT)
(expresión !)
* Da como resultado el valor lógico 1 si la expresion tiene un valor igual a cero.
* Da como resultado el valor lógico 0 si la expresión tiene un valor distinto de cero (línea **109**).

6.2 Ejemplo

Se define la dimensión de un vector mediante un entero; el concepto matriz se restringe, de momento, a un vector; se escribe un texto de salida mediante una sentencia lógica:

```
#include <iostream.h>
#include <fstream.h>

#define MAX_SIZE 25

int main (void)
{
 int size;
 ofstream MyReport("resultado.txt");

 cout << “Introduzca la dimensión del vector"<<endl;
 cin >> size;
 if (size > MAX_SIZE)
       {
        cerr << "Error, tamaño demasiado grande" << endl;
        MyReport << " *********************************** inicio" << endl;
        MyReport << " ERROR: dimension del vector debe ser menor de "
                 << MAX_SIZE   << endl;
        MyReport << " ************************************** fin" << endl;
        MyReport.close();
        return 1;
       }

 cout << "Vamos a trabajar con un vector de " << size
      << " componentes" <<endl;

 MyReport << " *********************************** inicio" << endl;
 MyReport << " Vector de " << size << " componentes" << endl;
 MyReport << " ************************************** fin" << endl;
 MyReport.close();

 return 0;
}
```

7 Ampliación de conceptos

7.1 Anidamientos de sentencias if

Se dice que una sentencia *if-else* está anidada cuando dentro de su formato general se encuentra otras sentencias *if*. Esto es;

```
if (condición 1)
{
   if (condición 2)  sentencia 1;
}
else
 sentencia 2;
```

Las llaves del código anterior indican que la cláusula *else* está emparejada con el primer *if.*

Cuando en el código de un programa aparecen sentencias *if-else* anidadas, y no existen llaves que definan claramente cómo emparejar los *else* y los *if*, la regla que se debe utilizar es que cada *else* corresponde con el *if* más próximo que no haya sido emparejado. Ejemplo:

```
if (condición 1)
   if (condición 2)
      sentencia 1;
   else
      sentencia 2;
```

En el código anterior, la cláusula *else* corresponde al segundo *if.*

Como es fácilmente comprensible, el anidamiento conduce a una lectura difícil del código. La manera de mantener legible la estructura lógica es utilizar las cláusulas y el sangrado para señalar cada bloque. Comparando los dos códigos inferiores, y mediante el sangrado, se puede seguir de forma clara qué está realizando el código.

```
if (condición 1)
  {
    if (condición 2)
        {sentencia 1;}
    else
        {sentencia 2;}
  }
```

```
if (condición 1)
    {
      if (condición 2)
          {sentencia 1;}
    }
else
    {sentencia 2;}
```

Ficha 4b : Control de flujo. Condicionales II

1 Objetivos generales

- Conocer las sentencias condicionales que permiten tomar decisiones (*switch* / *break*).

2 Código de trabajo (switch / break)

```
//Ficha 4b
/*Sentencia tipo switch*/

#include <iostream.h>

int main (void)
{
int i=1;
switch (i)
{
case 1:
cout <<"i es 1"<<endl;
break;

case 2:
cout <<"i es 2"<<endl;
break;

case 25:
 cout << “i es 25‟ << endl;
default:
cout <<"i no es ni 1 ni 2"<<endl;
break;
}
return 0;
}
```

3 Conceptos

3.1 Sentencias tipo switch-break

La sentencia *switch* permite ejecutar una de varias acciones, en función del valor de una expresión. Es una sentencia muy útil para decisiones múltiples.

La sintaxis de esta sentencia es la siguiente:

*switch (*expresión)
{
case expresión-constante *1:*
sentencia 1;
break;

case expresión-constante 2:
sentencia 2;
break;

case expresión-constante 3:
case expresión-constante 4:
sentencia 3;
break;

default:
sentencia n;
break;
}

La sentencia *switch* evalúa la expresión entre paréntesis y compara su valor con las constantes de cada *case***.** La ejecución de las sentencias comienza en el *case* cuya constante coincida con el valor de la expresión y continúa hasta el final del bloque de la sentencia *switch* o hasta que encuentra una sentencia tipo *break* o *return* que la saque del bloque.

En el código de trabajo expuesto, el bloque de la sentencia *switch* está formado por las sentencias existentes entre las líneas **007** y **020**. En la línea de código **007** se evalúa el valor de la variable *i* (el cual tal como se indica en la línea **006** tiene un valor asignado de 1) y se compara con el valor de las constantes de cada *case*. Como su valor coincide con el *case 1*, se imprime por pantalla el mensaje “i es 1” (línea **010**), y a continuación, al encontrarse en la línea **011** con la palabra *break*, sale del bloque y pasa a ejecutar la línea **021**.

Si no existe una constante igual al valor de la expresión, se ejecutan las sentencias que están a continuación de *default* (la cláusula *default* puede colocarse en cualquier parte del bloque *switch*, no necesariamente al final). Es decir, si *i* no es ni 1, ni 2, ni 25, se ejecuta desde la línea **017** hasta la **019**. Si *i* fuera 25, se ejecutaría a partir de la línea **015** y, al no hallar un *break*, continuaría hasta la sentencia **019** incluyendo al *default*.

La cláusula *break* al final del bloque *switch* no es necesaria, pero es aconsejable utilizarla para obtener una programación más clara.

4 Ejercicios (switch / break)

4.1 El programa pregunta dos números, qué operación deseas hacer, y muestra el resultado.

Una solución al ejercicio realizando la operación de suma o producto con dos números reales. De forma parecida a la vista en la ficha anterior pero ahora trabajando con las nuevas sentencias *switch* y *break*.

```
// Ficha 4b
/* el programa pregunta dos números, que operación deseas hacer
y muestra el resultado*/

#include <iostream.h>
int main (void)
{
double a, b;  int c;
cout << "Introduzca el primer número" <<endl ;                          cin >> a;
cout << "Introduzca el segundo número" <<endl ;                         cin >> b;
cout << "Que operación deseas hacer, 1(suma), 2(producto)" <<endl ;     cin >> c;

switch (c)
{
case 1:
cout <<"el resultado es:"<<a+b<<endl;
break;

default:
cout <<"el resultado es:"<<a*b<<endl;
break;
}
return 0 ;
}
```

4.2 Calcular áreas de diferentes polígonos

Una solución al ejercicio

```
//Ficha 4b
/* Calcular áreas de diferentes polígonos*/

#include <iostream.h>

int main (void)
{
double a, b, Area;  int figura;

cout<<figura = rectángulo(1), triángulo(2)"<<endl;
cout<<"¿que área quieres calcular?"<<endl;              cin>>figura;

switch (figura)
{
case 1:
cout <<"valor del lado menor ="; cin >>a;
cout <<"valor del lado mayor ="; cin >>b;               Area=a*b;
break;

case 2:
cout <<"valor de la base =";   cin >>a;
cout <<"valor de la altura ="; cin >>b;                 Area=a*b/2.0;
break;

default:
cout <<"figura equivocada"<<endl;                       Area=0;
break;
}
if (Area) cout<<"El área de la figura es="<<Area<<endl;
else cerr <<"error, la figura no tiene area"<<endl;
return 0 ;
}
```

Ficha 5a : Control de flujo. Bucles I

1 Objetivos generales

- Conocer las sentencias que permiten realizar bucles.

2 Código de trabajo (for)

```
//Ficha 5a
/* bucles for */

# include <iostream.h>

int main (void)
{
for (int i=2; i<4; i++)
cout <<"escribo el"<<i<<endl;
return 0;
}
```

3 Conceptos

3.1 Bucles for

La sentencia *for* permite ejecutar una instrucción, o un conjunto de ellas, un cierto número de veces deseado.

La sintaxis de esta sentencia es la siguiente:

```
for (variable = valor ; condición ; progresión-condición)
    {
        sentencia 1;
        sentencia 2;
        ...
    }
```

En el código de trabajo se utiliza la sentencia *for* para escribir en pantalla varias veces un mensaje.

La ejecución de la sentencia *for* se realiza :

1º) Inicializando la variable.

En la línea **006** se inicializa la variable *i* asignándole el número 2.

2º) Comprobando la condición.

En la línea **006** se comprueba inicialmente la condición 2<4.

Si la condición es verdadera (valor distinto de cero), se ejecutan las sentencias y se da un nuevo valor a la variable según lo indicado en la progresión-condición. En el código de trabajo, al cumplirse inicialmente la condición, se ejecuta la sentencia que imprime por pantalla el mensaje “escribo el 2” (línea **007**).

3) ejecutando la progresión-condición

A continuación se aumenta el valor de *i* en una unidad (i=3), se comprueba la condición 3<4, y se escribe por pantalla el mensaje “escribo el 3”; y así sucesivamente hasta que no se cumpla la condición, en cuyo caso se da por finalizada la ejecución de la sentencia *for*. En concreto, se vuelve a aumentar el valor de *i* en una unidad (i=4), como la condición 4<4 es falsa (valor igual a cero) se da por finalizado el bucle *for* y se pasa el control a la siguiente sentencia del programa. En consecuencia, no se ejecuta la sentencia de la línea **007** y se finaliza el programa.

4 Código de trabajo (continue / break)

```
101    //Ficha 5a
102    /*Salir o modificar un bucle for */

103    #include <iostream.h>
104    int main (void)
105    {
106    for (int i=1; i<10; i++)
107    {
108    if (i==2 || i==4) continue ;
109    if (i ==7) break;
110    cout <<"escribo el"<<i<<endl;
111    }
112    return 0;
113    }
```

5 Conceptos

5.1 Salir o modificar un bucle for

La sentencia *continue*, línea **108**, detiene la ejecución y vuelve al bucle *for* para comenzar una nueva iteración. La sentencia *break* (**109**), como ya se indicó en la ficha 4b, permite salir del bucle *for* sin ejecutar el resto de sentencias.

Entre las líneas **106** y **111** del código de trabajo anterior se ha programado una sentencia *for*. Esta sentencia inicializa la variable *i* a 1, comprueba la condición (i<10) y pasa a ejecutar las sentencias comprendidas entre llaves; en la línea **108** se comprueba la condición de que la variable *i* valga 2 ó 4, como esto no es así, no se ejecuta la cláusula *continue*, pasándose a ejecutar la sentencia 109; en ésta

se comprueba la condición de *i=7*, que al ser falsa hace que no se ejecute la sentencia *break*. Seguidamente se pasa a ejecutar la sentencia **110,** escribiéndose por pantalla el mensaje "*escribo el 1*". A continuación se aumenta el valor de la variable *i* en una unidad y, después de comprobar la condición i<10, se sigue con el bucle .

Obsérvese que en la iteración en la que la variable *i* sea 2 ó 4 se ejecutará la cláusula *continue* de la línea de código **108**, no ejecutándose entonces las sentencias **109** y **110** y volviendo a comenzar una nueva iteración en la línea **106**. En consecuencia, por pantalla no se escribirán los mensajes "escribo el 2" y "escribo el 4". Cuando la variable *i* tenga asignado el valor 7 en la línea **109,** se ejecutará la cláusula *break* saliendo del bucle *for* y pasando a la línea de código **112**, luego por pantalla tampoco aparecerán los mensajes "escribo el 7", "escribo el 8" ni "escribo el 9".

6 Ejercicios (for)

6.1 Escribir la tabla de multiplicar de un cierto número.

Solución propuesta al ejercicio.

```
// Ficha 5a
/* Escribir la tabla de multiplicar de un cierto número*/

#include <iostream.h>
int main (void)
{
int n;
cout << "Introduzca el número cuya tabla de multiplicar desea:"; cin >> n;
if (n==0)
cout << "la tabla es nula" <<endl ;
else
{
cout << "la tabla de multiplicar del número"<<n<<"es:"<<endl;
for (int i=1; i<10;i++)
cout <<i<<"*"<<n<<"="<<i*n<<endl;
}
return 0 ;
}
```

6.2 Listar los 20 primeros números pares/impares en orden decreciente.

Una posible solución al ejercicio.

```
//Ficha 5a
/* Listar los 20 primeros números pares/impares en orden decreciente*/

#include <iostream.h>

void main (void)
{
int n;

cout<<"Los 20 primero números pares en orden decreciente"<<endl;
for (int i=0; i<21; i+=2) { n=20; n-=i; cout <<"numero par:"<<n<<endl; }

cout<<"Los 20 primero números impares en orden decreciente"<<endl;
for (i=19; i>0; i-=2) cout <<"numero impar:"<<i<<endl;
}
```

7 Ejemplo

Se define la dimensión de un vector mediante un entero; el concepto matriz se restringe, de momento, a un vector; se escribe un texto de salida mediante una sentencia lógica y se inicializa el vector/matriz mediante un bucle *for*. Atención porque la sintaxis Matrix [algo] con corchetes se debe tomar como una definición, en fichas posteriores se aclarará el funcionamieto del operador []:

```
#include <iostream.h>
#include <fstream.h>

#define MAX_SIZE 25

int main (void)
{
 int Matrix [MAX_SIZE];
      // esto es un vector, se ve con detalle en las fichas 12
      // de momento lo tomamos como una definición para poder trabajar
      // se está definiendo un vector Matrix[25] de 25 componentes

 int size;
 ofstream MyReport("resultado.txt");

 cout << "Introduzca la dimensión del vector"<<endl;
 cin >> size;
 if (size > MAX_SIZE)
{
 cerr << "Error, tamaño demasiado grande" << endl;
 MyReport << " *********************************** inicio"
          << endl;
 MyReport << " ERROR: dimension del vector debe ser menor de "
          << MAX_SIZE   << endl;
 MyReport << " ************************************** fin"
          << endl;
 MyReport.close();
 return 1;
}

 for (int i=0; i<size; i++) Matrix[i]=0;
                                 // inicializando a cero cada componente
 cout << "Vamos a trabajar con un vector de " << size
      << " componentes" <<endl;

 MyReport << " *********************************** inicio" << endl;
 MyReport << " Vector de " << size << " componentes" << endl;
 i=0;
 do { MyReport << Matrix[i] << " "; } while (++i<size);
                   // atencion a la precedencia de operadores
                   // el resultado es distinto si while (i++<size)
 MyReport << endl;
 MyReport << " ************************************** fin" << endl;
 MyReport.close();

 return 0;
}
```

Ficha 5b : Control de flujo. Bucles II

1 Objetivos generales

- Conocer las sentencias que permiten realizar bucles.

2 Código de trabajo (do - while)

```
//Ficha 5b
/* Sentencias tipo do-while */

#include <iostream.h>
int main (void)
{
int i=0; j=6;
do
{
cout<<"el número 6 es mayor que el"<<i<<endl;
}
while (i++<j);
return 0;
}
```

3 Conceptos

3.1 Sentencias tipo do-while

La sentencia *do-while* ejecuta una sentencia (simple o compuesta) una o más veces, dependiendo del valor de una expresión.

Su sintaxis es :

do
{
   sentencias ;
}
while (condición);

Nótese que la sentencia *do-while* finaliza con un **;**

La ejecución de esta estructura se ejecuta de la siguiente forma:

1. Se ejecuta la sentencia. En el código de trabajo descrito, se ha utilizado una estructura *do-while* de forma que se inicia la ejecución en la línea **009** imprimiéndose por pantalla el mensaje "el número 6 es mayor que el 0".

2. Se evalúa la condición. Si el resultado es falso (valor igual a cero), se pasa el control a la siguiente sentencia del programa. Si el resultado es verdadero (valor distinto de cero) el proceso se repite desde el principio.

En el código de trabajo se evalúa la condición i++<j (atención a la precedencia de operadores), es decir se aumenta *i* en una unidad (i=1) y se comprueba si es menor que *j* (1<6). Al ser verdadera la condición, se ejecuta de nuevo la línea **009,** escribiéndose por pantalla el mensaje " el número 6 es mayor que el 1", y así sucesivamente hasta que la condición de finalización no se cumpla. Esto ocurrirá cuando evaluemos 6<6, por lo que el último mensaje que se escribirá por pantalla será "el número 6 es mayor que el 5". Nótese que el bloque de sentencias se realiza por lo menos una vez, ya que la condición de finalización se evalúa al final.

4 Ejercicios (do - while)

4.1 Listar los 20 primeros números pares en orden decreciente.

Una solución al ejercicio.

```
//Ficha 5b
/* Listar los 20 primeros números pares en orden decreciente*/

#include <iostream.h>

void main (void)
{
cout<<"lista de los 20 primero números pares en orden decreciente"<<endl; int i=22;

do
{
i-=2; cout <<"numero par:"<<i<<endl;
}
while (i>0);
}
```

5 Código de trabajo (goto)

```
//Ficha 5b
/* Sentencia goto*/

#include <iostream.h>

int main (void)
{
goto jumping;
cout<<"aquí no llego"<<endl;
jumping:
cout <<"aquí si""<<endl;
return 0;
}
```

6 Conceptos

6.1 Sentencia goto

La sentencia *goto* permite saltar a una línea específica del programa, identificada por una etiqueta.

Su sintaxis es :

> *goto etiqueta*;
> .
> .
> *etiqueta*:
> sentencia;

En la línea **106** del código de trabajo se utiliza la sentencia *goto* para saltar a la línea **108,** ejecutándose la sentencia que escribe por pantalla el mensaje “aqui si” (línea **109**).

En general, se recomienda **no utilizar nunca** esta sentencia porque complica el seguimiento del código y va en contra de la filosofía de programación de objetos. Sin embargo, durante el período de pruebas del programa puede ser interesante utilizarla para encontrar errores y facilitar la depuración del código.

7 Código de trabajo (while)

```
//Ficha 5b
/* Sentencias tipo while */

#include <iostream.h>
int main (void)
{
int i=0;
while (i<6)
{
cout<<"el número 6 es mayor que el "<<i++<<endl;
}
return 0;
}
```

8 Conceptos

8.1 Sentencias tipo while

La sentencia *while* funciona de forma parecida al bucle *do-while*. La diferencia principal estriba en que el bucle *do-while* asegura que, al menos, se ejecuta una vez el código contenido entre las llaves; mientras que el *while* depende de la condición lógica.

Su sintaxis es :

> *while* (condición)
> {
>   sentencias;
> }

9 Ejercicios (while)

9.1 Listar los 20 primeros números pares en orden decreciente.

Posible solución al ejercicio.

```
//Ficha 5b
/* Listar los 20 primeros números pares en orden decreciente*/

#include <iostream.h>

int main (void)
{
cout<<"lista de los 20 primero números pares en orden decreciente"<<endl;

int i=22;

while (i>0)
{
cout <<"numero par: "<<i<<endl;
i-=2;
}

return 0 ;
}
```

10 Ejemplo

Se define la dimensión de un vector mediante un entero; el concepto matriz se restringe de momento a un vector; se escribe un texto de salida mediante una sentencia lógica y se inicializa el vector/matriz mediante un bucle *for*; finalmente se escribe el resultado con un bucle *do-while*:

```
#include <iostream.h>
#include <fstream.h>

#define MAX_SIZE 25

int main (void)
{
 int Matrix [MAX_SIZE];
      // esto es un vector, se ve con detalle en las fichas 12
      // de momento lo tomamos como una definición para poder trabajar
 int size;
 ofstream MyReport("resultado.txt");

 cout << "Introduzca la dimensión del vector"<<endl;
 cin >> size;
 if (size > MAX_SIZE)
{
 cerr << "Error, tamaño demasiado grande" << endl;
 MyReport << " *********************************** inicio"
          << endl;
 MyReport << " ERROR: dimension del vector debe ser menor de "
          << MAX_SIZE   << endl;
 MyReport << " ************************************** fin"
          << endl;
 MyReport.close();
 return 1;
}
```

```
 for (int i=0; i<size; i++) Matrix[i]=0;
                                 // inicializando a cero cada componente
 cout << "Vamos a trabajar con un vector de " << size
       << " componentes" <<endl;

 MyReport << " *********************************** inicio" << endl;
 MyReport << " Vector de " << size << " componentes" << endl;
 i=0;
 do { MyReport << Matrix[i] << " "; } while (++i<size);
                   // atencion a la precedencia de operadores
                   // el resultado es distinto si while (i++<size)
 MyReport << endl;
 MyReport << " ************************************** fin" << endl;
 MyReport.close();

 return 0;
}
```

Ficha 6: Funciones

1 Objetivos generales

- Conocer la motivación y sentido de las funciones.

2 Código de trabajo

```
// Ficha 6
/* funciones */

#include <iostream.h>

int checking (int i, int j);     //declaración función (prototipo)

int main (void)                  //programa principal
{
int i;
cout<< "escribe un numero"<<endl; cin>>i;
int j;
cout<<"escribe otro numero"<<endl;cin>>j;
if (!checking(i,j))              //llamada a la función
                           //(si la función retorna 0)
cout<<i<<" es mayor que "<<j<<endl;
else                             //(si la función retorna 1)
cout<<i<<" es menor que "<<j<<endl;
return 0;
}

int checking (int i, int j)      //definición de la función
                           //(implementación)
{
if (i<j) return 1;
return 0;
}
```

3 Conceptos

3.1 Qué son las funciones

Una *función* es una relación operacional entre varias entidades, se suele representar como una correspondencia entre unos parámetros de entrada y un resultado de salida. En el presente caso, una

función es un bloque de código que recibe unos valores de entrada (argumentos) y devuelve un resultado.

Las funciones sirven para *encapsular* las operaciones en pequeñas unidades que hacen que los programas sean más claros y más fiables. Mediante la compartimentación estanca del código mejora la calidad de la programación. El concepto de función es similar al de subrutina, para aquellos lectores que conozcan lenguajes como Fortran o Basic, pero con la particularidad del retorno de un valor y la estanqueidad del acceso a las variables.

Entre las líneas **017** a la **021** se define la función de forma separada del código principal, que está entre las líneas **005** a **016,** y sólo es llamada una vez, en la línea **011**.

3.2 Declaración de una función (prototipo)

Antes de escribir el código de la función, la función como tal debe declararse para que el compilador sepa que existe y cuáles son sus argumentos. En un archivo de cabezera *.h, o al principio del archivo, como es el caso de la línea **004**, la función debe *declararse.* La sintaxis de una declaración genérica es:

```
TipoRetorno    NombreFunción   ( TipoArgumento1  a , TipoArgumento2  b , ... ) ;
```

En las declaraciones de los argumentos debe ponerse el tipo (TipoArgumento1, TipoArgumento2 , etc.) y, opcionalmente, un nombre de variable identificativo (a, b, ...) que será desechado por el compilador. En consecuencia, no tiene por qué coincidir con el que se utilice posteriormente en la definición de dicha función, pero puede ayudar a identificar qué tipo de argumento se está pasando a la función. Por ejemplo, si se lee:

```
double CosteTotal (int unidades, double precio);
```

Queda bastante claro el objetivo de dicha función y qué papel juega cada argumento, bastante mejor que un simple:

```
double CosteTotal (int, double);
```

Si no se especifica el tipo de retorno, se asumirá *int* por defecto, aunque se recomienda especificarlo siempre por razones de claridad.

En la línea **004** del código de trabajo se ha declarado la función de nombre *checking*, cuyos argumentos de entrada son dos enteros y el retorno también otro número entero. Nótese que la sentencia va seguida de punto y coma para señalar que sólo es una declaración.

3.3 Definición de una función (implementación)

Definir una función es describir lo que hace. Una función sólo puede ser definida una vez y en cualquier parte del programa, excepto dentro de otra función. La sintaxis de la implementación de una función es:

```
TipoRetorno    NombreFunción   ( Tipo1  i , Tipo2  j , Tipo3  k )
{
Sentencia1;
...
SentenciaN;
}
```

Si algún argumento no tiene nombre no se utiliza.

En las líneas de código **017-021** se define la función de nombre *checking* que había sido declarada en la línea **004**. Los argumentos de la función, con sus nombres de variables, se usan para realizar una comparación lógica y devolver un valor 0 o 1.

3.4 Llamada a la función

En el código se debe llamar a una función por su nombre y pasarle las variables de los argumentos entre paréntesis (obligatorios aunque no tenga parámetros, o sea tipo *void*). Los parámetros que entran en la función deben ser del mismo tipo que los declarados en el prototipo y en la definición, en caso contrario se recomienda realizar una conversión (*casting*) para evitar errores de compilación y ejecución.

En la línea **011** se realiza una llamada a la función *checking*. La función retorna un 1 o un 0, por ello se puede utilizar el valor que devuelve para realizar una sentencia lógica con el operador ! (negación). De esta forma se ejecuta la sentencia de la línea **012** si devuelve un 0, y en caso contrario, se ejecuta la sentencia de la línea **014**.

3.5 Retorno de valores

La sentencia *return* es necesaria para poder retornar valores desde las funciones. Su sintaxis es :

```
return    expresión  ;
```

La *expresión* debe ser del mismo tipo o convertible al tipo de retorno de la función. Si no se pudiese convertir, el compilador daría un aviso (*warning*) o un error. Siempre es recomendable retornar un valor para poder verificar la correcta ejecución de la función; en este caso el retorno no está asociado a ninguna operación con los argumentos, sino con el correcto funcionamiento del programa.

En la línea de código **020** se utiliza la sentencia *return 0* para retornar un *int*, que es el tipo de retorno de la función *checking*, aunque según el resultado de la sentencia lógica puede retornar un 1 en la línea **019**.

Se pueden declarar funciones que no devuelven ningún valor, son declaradas como *void*. Por ello puede omitirse la sentencia *return* y, en caso de ponerla, se debe escribir sin la *expresión*.

4 Ejercicios

4.1 El programa pregunta dos números que operación quieres hacer y muestra el resultado.

En este caso se ha definido una solución parecida a la que se había planteado en la ficha de condicionales 5a. No obstante, las operaciones sobre los números se realizan a través de llamadas a funciones, y no mediante la aplicación directa de los operadores. Por supuesto, la alternativa escogida es más ineficiente, pero más ilustrativa del concepto y mecanismo de las funciones.

```
// Ficha 6
/* el programa pregunta dos números que operación quieres hacer y muestra el
resultado */

#include <iostream.h>

double Suma (double x, double y);       //declaración función Suma
double Producto (double x, double y);   //declaración función Producto

int main (void)     //programa principal
{
double a, b;
int c;
cout << "escribe un numero"<< endl; cin>>a;
cout << "escribe otro número"<<endl; cin>>b;
cout <<"Que operación deseas hacer, 1(suma), 2(producto)"<<endl;
cin>>c;

if (c==1)
cout <<"el resultado de la suma es:"<<Suma(a,b)<<endl;
else
cout<<"el resultado del producto es:"<<Producto(a,b)<<endl;
return 0;
}

double Suma (double x, double y) //definición función Suma
{
double z;
z=x+y;
return z;
}

double Producto (double x, double y)    //definición función Producto
{
return (x*y);
}
```

4.2 Calcular longitudes entre puntos.

Solución al ejercicio.

```
// Ficha 6
/* cálculo de la longitud entre dos puntos. */
#include <iostream.h>
#include <math.h> // es necesaria para la función sqrt ()

//declaración función

double longitud
 (double x0,double y0,double z0,double xf,double yf,double zf);

//programa principal

int main (void)
{
double x1,y1,z1; double x2,y2,z2;
cout<<"coordenada x del primer punto"; cin>>x1;
cout<<"coordenada y del primer punto"; cin>>y1;
cout<<"coordenada z del primer punto"; cin>>z1;
cout<<"coordenadas x y z del segundo punto"; cin>>x2>>y2>>z2;

//llamada funcion

cout <<" la longitud es:"<<longitud (x1,y1,z1,x2,y2,z2)<<endl;
```

```
return 0;
}
//definición función

double longitud
 (double x0,double y0,double z0,double xf,double yf,double zf)
{
return sqrt ((((xf-x0)*(xf-x0))+((yf-y0)*(yf-y0))+((zf-z0)*(zf-z0))));
}
```

4.3 Calcular áreas de polígonos.

Solución al ejercicio para los cuadrados, rectángulos y triángulos. Nótese cómo una única función sirve para calcular tanto los rectángulos como los cuadrados.

```
// Ficha 6
/* cálculo áreas de polígonos */
#include <iostream.h>

//declaración de funciónes
int  AreaCuadrilateros  (double lado_a, double lado_b);
int  AreaTriangulos  (double  lado_a, double  altura_b);

//programa principal

int main (void)
{
double a,b;  int figura;
cout>>"Que figura quiere analizar,
       Cuadrado (1), Rectangulo(2), Triangulo(3):"<<endl;
cin>>figura;

switch (figura)
{
case 1:
cout<<" introduzca el valor del lado"<< endl;
cin>> a;
if (!AreaCuadrilateros(a,a))     //llamada función
cout<<"ojo el area es negativa "<<endl;       break;

case 2:
cout<<" introduzca el valor del lado a  y b:"<< endl;
cin>> a, b;
if (!AreaCuadrilateros(a,b))     //llamada función
cout<<"ojo el area es negativa "<<endl;
break;

case 3:
cout<<" introduzca la base y la altura:"<< endl;
cin>> a, b;
if (!AreaTriangulos(a,b))        //llamada función
cout<<"ojo el area es negativa "<<endl;
break;

default:
cout<<"figura equivocada"<<endl;
break;
}
return 0;
}
```

```
//definición de las funciones

int AreaCuadrilateros (double  lado_a, double lado_b)
{
double A;
A=lado_a * lado_b;
if (A<0) return 0;
else cout <<"el area es:"<<A<<endl;
return 1;
}

double AreaTriangulos (double  lado_a, double altura_b)
{
double A;
A=lado_a * altura_b / 2.;
if (A<0) return 0;
else cout <<"el area es:"<<A<<endl;
return 1;
}
```

5 Ejemplo

Se reescribe el programa de la ficha anterior, pero agrupando el código en funciones:

```
#include <iostream.h>
#include <fstream.h>

#define MAX_SIZE 25

void report_error(void);
int report_final (int dimen, int matriz[]);

int main (void)
{
 int size;
 int Matrix [MAX_SIZE];
      // esto es un vector, se ve con detalle en las fichas 12
      // de momento lo tomamos como una definición para poder trabajar

 cout << "Introduzca la dimensión del vector"<<endl;
 cin >> size;
 if (size > MAX_SIZE) report_error();

 for (int i=0; i<size; i++) Matrix[i]=0;
                                // inicializando a cero cada componente
 if (report_final(size,Matrix))
      cout << "Vamos a trabajar con un vector de " << size
              << " componentes" <<endl;
 else cerr << "error general" << endl;

return 0;
}

void report_error(void)
{
 ofstream MyReport("resultado.txt");
 cerr << "Error, tamaño demasiado grande" << endl;
 MyReport << " ********************************* inicio"
          << endl;
 MyReport << " ERROR: dimension del vector debe ser menor de "
          << MAX_SIZE   << endl;
 MyReport << " ************************************ fin"
```

```
         << endl;
 MyReport.close();
 return;
}

int report_final (int dimension, int matriz[])
{
 ofstream MyReport("resultado.txt");
 MyReport << " *********************************** inicio" << endl;
 MyReport << " Vector de " << dimension << " componentes" << endl;
 int i=0;
 do { MyReport << matriz[i] << " "; } while (++i<dimension);
                    // atencion a la precedencia de operadores
                    // el resultado es distinto si while (i++<dimension)
 MyReport << endl;
 MyReport << " ************************************** fin" << endl;
 MyReport.close();
 return 1;
}
```

Ficha 7: Ámbito de variables

1 Objetivos generales

- Conocer el concepto de ámbito de las variables.

2 Código de trabajo

```
// Ficha 7
/* Ambito de variables. Variables locales y globales */

#include <iostream.h>

int a_global=1;                 //variable global

void write_var(int a);          //declaración función vrite_var

int main (void)                 //programa principal
{
int a_local=2.;                 //variable local
write_var(a_local);             //llamada a la función write_var
return 0;
}

void write_var(int a)           //definición función write_var
{
int a_local=3.;                 //variable local
cout<<"el valor de la variable que pasamos es="<<a<<endl;
cout<<"el valor de la variable local es="<<a_local<<endl;
cout<<"el valor de la variable global es="<<a_global<<endl;
return ;
}
```

3 Conceptos

3.1 Ámbito o alcance (*scope*) de variables

Las variables son una entidad fundamental en cualquier programa, sus valores deciden el resultado de la ejecución del programa. El cambio en los valores de una variable se produce en dos casos: el código la modifica conforme a las necesidades del programa, o bien, se producen errores en la gestión de memoria por parte del sistema operativo o de ejecución del programa. El primer caso es el deseable, el

segundo el impredecible. Hay que aceptar que las variables son entidades frágiles, sólo existen en un lugar de la memoria del ordenador por un espacio reducido de tiempo, por ello hay que cuidarlas al máximo.

El riesgo de que se produzca un cambio en una variable de forma involuntaria aumenta de acuerdo con el ámbito de la variable. El *ámbito* o alcance de una variable es el lugar donde ha sido declarada, ese espacio de código en el que existe y desde el que puede ser modificada. Por ejemplo, la variable *a_global* definida en la línea **004** tiene su alcance desde la línea de definición hasta la última (**019**). En cambio la variable *a_local* de la línea **008** muere en la línea **011**. Posteriormente, ya se verá por qué esto es así, pero démonos cuenta que la *a_global* es más frágil que la *a_local* porque se puede acceder a ella desde cualquier parte del código y en cualquier momento durante el tiempo de ejecución.

Es lógico pensar que cuantas más variables maneje el programa y cuanto mayor sea su ámbito, más probabilidades habrá que los valores en memoria puedan variar bruscamente por errores de ejecución o de gestión de memoria. Sobre todo si el código crece por la colaboración de varios programadores. En general, es deseable que las variables tengan la vida lo más corta posible y que sólo se puedan modificar desde pequeños bloques de código.

El alcance en el que pueden estar definidas y, en consecuencia, ser utilizadas las variables son:

3.1.1 Globales
Toda variable declarada fuera de bloques tiene un ámbito global, por ejemplo la variable de la línea **004**. Esto significa que puede ser modificada desde cualquier punto del programa. Por lo tanto, su gran ventaja es al mismo tiempo su mayor inconveniente. Utilizar variables globales va en contra del encapsulamiento, en consecuencia se deben evitar en lo posible.

Si la declaración de una variable global (línea **004**) es anterior a un ámbito local (líneas **007-010**), puede accederse a ellas desde este campo local. Si no es así, para el acceso desde un campo local debe utilizarse la palabra *extern*. Esto es especialmente inconveniente cuando se trabaja con diferentes archivos de código. Para evitar el abuso de la etiqueta *extern* se recomienda definir las variable de tipo global en un archivo de cabezera (*.h) e incorporarlo mediante la sentencia *#include*.

Se pueden definir variables con el mismo nombre, del mismo o distinto tipo, siempre que no se hayan definido en el mismo campo (líneas **008** y **014**).

Atención porque, al definir con el mismo nombre una variable local y una global, tiene predominancia el valor de la local en el trozo en el que actúa.

3.1.2 Locales
Están delimitados por llaves { y } (no por paréntesis), ya sea de bloque o de función.

Bloque: Si declaramos una variable dentro de un bloque {} (línea **008**), la variable sólo será válida dentro de ese bloque (línea **009**) y no será accesible desde fuera.

Función: Si declaramos una variable dentro de una función (línea **014**), la variable puede ser utilizada en cualquier sitio de la función donde ha sido declarada (línea **016**). Las variables que se pasan en los argumentos de la función siempre se copian localmente, esto significa que se duplican en memoria y que no se modifica su valor al salir de la función. Para alterar sus valores se deberán pasar como apuntadores o referencias, esta propiedad se verá en fichas posteriores.

3.1.3 Clases
Este campo se verá en la ficha 9

3.1.4 Declaración (prototipo)

Los nombres que se les da a las variables en la lista de argumentos de la definición de una función son identificadores inútiles y son olvidados por el ordenador. En la línea de código **005** se declara la función de nombre *write_var* que tiene un argumento de entrada de tipo *int* al que se le ha dado el nombre *a* (podía haberse omitido este identificador) y un retorno *void*.

4 Ejercicios

4.1 Modificar el código anterior y pasar variables globales, locales, etc.

4.1.1 Solución al ejercicio, versión primera

```
// Ficha 7
/* Ambito de variables. Variables locales y globales. Modificación 1 */
#include <iostream.h>

int a_global=5;           //declaración variable global , valor 5
void write_var(int a);    //declaración función

int main (void)           //programa principal
{
int a_local=2.;           //declaración variable local, valor 2
write_var(a_global);      //llamamos a la función con argumento entero 5
return 0;
}

void write_var(int a)     //definición función
{
int a_local=3.;
cout<< "el valor de la variable que pasamos es="<<a<<endl;
cout<<"el valor de la variable local es="<<a_local<<endl;
return ;
}
```

4.1.1 Solución al ejercicio, versión segunda

```
// Ficha 7
/* Ambito de variables. Variables locales y globales. Modificación 2 */
#include <iostream.h>

int a_global=5;           //declaración variable global, valor 5
void write_var(int a);    //declaración función

int main (void)           //programa principal
{
int a_global=6;           //accedemos a la variable global
                          //desde un ámbito local, valor 6
write_var(a_global);      //llamamos a la función con argumento entero 6
return 0;
}

void write_var(int a)     //definición función
{
int a_local=3.;
cout<< "el valor de la variable que pasamos es="<<a<<endl;
cout<<"el valor de la variable local es="<<a_local<<endl;
return ;
}
```

4.2 Definir alguna otra función para jugar con los conceptos de ámbito.

Solución al ejercicio

```
// Ficha 7
/* Introducir una nueva función para practicar
con los conceptos de ámbito local y global */

#include <iostream.h>

int a_global=1;                   //variable global con valor 1

void write_var(int a);            //declaración función write_var
void checking (int i,int j);      //declaración función checking

int main (void)                   //programa principal
{
int a_local=2.;                   //variable local con valor 2
write_var(a_local);               //llamada a función
                                  //con argumento de valor  2
checking (a_local, a_global);     //llamada a función con argumentos 2 y 1
return 0;
}

void write_var(int a)             //definición función write_var
{
int a_local=3.;
cout<<"el valor de la variable que pasamos es="<<a<<endl; //el valor=2
cout<<"el valor de la variable local es="<<a_local<<endl; //el valor=3
return ;
}
void checking (int i, int j)      //definición función checking
{
if (i>j) cout<<i<<" es mayor que "<<j<<endl;  //2 es mayor que 1
else cout <<i<<" es menor que  "<<j<<endl;
return;
}
```

5 Ejemplo

El código hace lo mismo que en la ficha anterior, pero el tratamiento de las variables es distinto; se recomienda comparar éste con el ejemplo de la ficha anterior:

```
#include <iostream.h>
#include <fstream.h>

#define MAX_SIZE 25

ofstream MyReport("resultado.txt"); // variable global
int size;                           // variable global

void report_error(void);
int report_final (int matriz[]);       // ahora sólo pasamos los datos

int main (void)
{
 int Matrix [MAX_SIZE];  // variable local en main
      // esto es un vector, se ve con detalle en las fichas 12
      // de momento lo tomamos como una definición para poder trabajar
```

```
 cout << "Introduzca la dimensión del vector"<<endl;
 cin >> size;  // como es global no hace falta definirla
 if (size > MAX_SIZE) report_error();

 for (int i=0; i<size; i++) Matrix[i]=0;
                                 // inicializando a cero cada componente
 if (report_final(Matrix))
      cout << "Vamos a trabajar con un vector de " << size
              << " componentes" <<endl;
 else cerr << "error general" << endl;

return 0;
}

void report_error(void)
{
 cerr << "Error, tamaño demasiado grande" << endl;
 MyReport << " ********************************** inicio"
          << endl;
 MyReport << " ERROR: dimension del vector debe ser menor de "
          << MAX_SIZE   << endl;
 MyReport << " ************************************** fin"
         << endl;
 MyReport.close();
 return;
}

int report_final (int matriz[])
{
 MyReport << " ********************************** inicio" << endl;
 MyReport << " Vector de " << size << " componentes" << endl;

      // De donde sale size !!! el ejemplo es pequeño y fácil de seguir
      // ¿Puede imaginar que pasa en un código más grande?
      // Nunca siga la estrategia del código que acaba de leer
// el ejemplo de la ficha anterior es mucho más recomendable !!!

 int i=0;
 do { MyReport << matriz[i] << " "; } while (++i<size);
                   // atencion a la precedencia de operadores
                   // el resultado es distinto si while (i++<size)
 MyReport << endl;
 MyReport << " ************************************** fin" << endl;
 MyReport.close();
 return 1;
}
```

Finalmente, al código de la ficha anterior se le añade la de suma de matrices dentro del *main*:

```
#include <iostream.h>
#include <fstream.h>

#define MAX_SIZE 25

void report_error(void);
int report_final (int dimen, int matriz[]);

int main (void)
{
 int size;
 int A [MAX_SIZE], B[MAX_SIZE], C[MAX_SIZE];
      // esto es un vector, se ve con detalle en las fichas 12
      // de momento lo tomamos como una definición para poder trabajar
```

```
 cout << "Introduzca la dimensión del vector"<<endl;
 cin >> size;
 if (size > MAX_SIZE) report_error();

 for (int i=0; i<size; i++) {A[i]=i; B[i]=i; C[i]=0;}
                                 // inicializando cada componente
 for (int i=0; i<size; i++) {C[i]=A[i]+B[i];}
                                 // realizamos la suma
 if (report_final(size,C))
      cout << "Vamos a trabajar con un vector de " << size
             << " componentes" <<endl;
 else cerr << "error general" << endl;

return 0;
}

void report_error(void)
{
 ofstream MyReport("resultado.txt");
 cerr << "Error, tamaño demasiado grande" << endl;
 MyReport << " *********************************** inicio"
          << endl;
 MyReport << " ERROR: dimension del vector debe ser menor de "
          << MAX_SIZE   << endl;
 MyReport << " ************************************** fin"
         << endl;
 MyReport.close();
 return;
}

int report_final (int dimension, int matriz[])
{
 ofstream MyReport("resultado.txt");
 MyReport << " *********************************** inicio" << endl;
 MyReport << " Vector de " << dimension << " componentes" << endl;
 int i=0;
 do { MyReport << matriz[i] << " "; } while (++i<dimension);
                    // atencion a la precedencia de operadores
                    // el resultado es distinto si while (i++<dimension)
 MyReport << endl;
 MyReport << " ************************************** fin" << endl;
 MyReport.close();
 return 1;
}
```

6 Ampliación de conceptos

6.1 Variables estáticas

Son variables que retienen sus valores a lo largo de la ejecución de todo el programa. Todas las variables globales son estáticas por defecto; no obstante, las variables locales se pueden hacer estáticas con el uso del identificador *static*. Esto quiere decir que su valor se conserva entre sucesivas llamadas a la función. Hasta aquí es similar a una variable global, pero la ventaja es que no es accesible desde fuera, con lo que se cumple el principio de encapsulación de datos. Las variables estáticas se inicializan a cero (convertido al tipo apropiado) si no se les da un valor en la definición. Estas variables son muy útiles cuando se trabaja con recursividad. Sin embargo, en el interior de las clases comparten el mismo espacio para todos los objetos y esto puede dar lugar a problemas.

Ejemplo:

```
#include <iostream.h>

void Saldo (double euros);

int main (void)
{
Saldo (1000);
Saldo (-1500);
return 1;
}

void Saldo (double euros)
{
static double Mi_Dinero ;
Mi_Dinero += euros;
cout<<"tengo en cuenta:"<<Mi_Dinero<<"euros"<<endl;
}
```

En el ejemplo anterior se ha definido una función denominada *Saldo*, a la cual se accede mediante un argumento de tipo *double,* que representa la cantidad de dinero que extraemos (-) o ingresamos (+) en cuenta. En el ámbito local de definición de esta función se ha utilizado una variable estática, con lo cual su valor se conserva para sucesivas llamadas de la función.

Ficha 8: Abstracción de datos

1 Introducción

Se dice que la *abstracción* es la acción de aislar, de considerar en la mente, una parte como algo separado de un todo.

La capacidad de abstracción está íntimamente relacionada con los mecanismos del razonamiento humano. La abstracción categoriza la realidad, por ejemplo en la siguiente línea aparecen muchos números aparentemente inconexos:

1, 2, 3, 2.5, -33, -12.3

Este conjunto de números se puede clasificar en función de distintos criterios, por ejemplo podemos distinguir dos categorías de números, los positivos y negativos. En el concepto abstracto de número positivo encontraríamos: 1, 2, 3, 2.5, mientras que en el de negativo estarían el -33 y el -12.3.

Además, nuestra mente, en función de los conocimientos de que dispongamos, nos permite fijar otra organización de la información, por ejemplo números enteros y números reales. En este caso, la agrupación de los números en categorías sería distinta: 1, 2, 3, -33 y 2.5, -12.3.

En este ejemplo tan sencillo se han introducido los elementos básicos que se desarrollarán a continuación. Nótese que en primer lugar se cuenta con una información, una simple lista de números, en segundo lugar se define un criterio para organizar la información basado en los conceptos abstractos de números positivos y negativos, o de forma alternativa, de enteros y reales. Y finalmente se organiza dicha información dividiendo los datos en dos grupos. Esto tan sencillo marca de forma importante la filosofía de la programación.

Hasta ahora el programador ha utilizado los tipos de datos que suministra el lenguaje por defecto. En consecuencia, la programación ha conllevado un esfuerzo, adaptar aquello que quiere decirse a las capacidades del lenguaje. Para los propósitos de la programación resulta interesante trabajar con datos que representen conceptos abstractos, de esta manera el programador deberá pensar de forma natural, de acuerdo con el pensamiento humano. Es decir, si se debe trabajar con variables que contengan el concepto empleado, debe existir un tipo de dato *empleado* y no un *string* nombre, número de DNI, etc.

En definitiva, la abstracción de datos pretende acercar la programación al concepto del lenguaje natural, el lenguaje del pensamiento.

2 Objetivos generales

- Ver como la abstracción de datos acerca el lenguaje de programación al lenguaje natural y qué beneficios comporta en la escritura de un código claro y reutilizable.

3 Código de trabajo

```
//Ficha 8
/*Definición tipo de datos. Estructuras */

#include <iostream,.h>

struct complex
{
double real, imag;
};

int main (void)
{
struct complex a;
a.real =1;
a.imag =1.;
cout<<complex="<<a.real<<"+"<<a.imag<<"i"<<endl;
return 0;
}
```

4 Conceptos

4.1 Cómo definir tipos de datos

Como en otros lenguajes, un tipo de datos es un concepto con el que se designa un conjunto de datos que tienen las mismas características, por ejemplo 2 es un tipo *int*, 1.5 es un tipo *double*, etc.

A partir de los tipos de datos básicos explicados en la ficha 2 y que son aquellos que suministra el lenguaje de programación por defecto, se pueden definir otros más complejos agrupando un conjunto de variables (miembros) en un nuevo tipo. C++ permite dos agrupaciones: Estructuras y Clases. Las estructuras son simples agrupaciones de datos; en cambio, las clases además incorporan funciones para manipular los datos. Las clases son elementos básicos del lenguaje C++, de naturaleza más compleja; se analizarán en fichas posteriores.

4.2 Estructuras

Las estructuras son agrupaciones de variables. De manera que las diferentes variables juntas crean un nuevo tipo de dato. Su sintaxis es la siguiente:

```
struct  nombre
{
   tipo1   Variable1;
   tipo2   Variable2;
} Var1, Var2;
```

donde nombre,Var1 y Var2 son opcionales; si no se ponen, se declara la estructura y su nombre; si se ponen, se declara la estructura y además variables del tipo de la estructura.

Si se desea trabajar con números complejos, lo más lógico es pensar en un *double* que represente la parte real y otro que represente la parte imaginaria; por lo tanto, en código se debería tener una sintaxis como esta:

double r, i;

Por supuesto, si hay que realizar muchas operaciones y se tienen muchas variables, será fácil perder la pista de ambas variables. Por ello, ¿no sería más deseable poder escribir algo como esto?:

complex a;

Donde *a* representa un número complejo que puede sumarse, restarse, etc. En consecuencia, el programador sólo debe preocuparse de una única variable *a*, en lugar de *r* e *i*. Nótese que se está definiendo un nuevo tipo de dato, *complex*, que no está definido en el lenguaje por defecto; es tarea del programador definir este tipo.

En el código de trabajo se define un tipo de dato de nombre *complex* (línea **004**), compuesto por dos miembros *doubles* (línea **006**) que representan respectivamente la parte real y la parte imaginaria de un número complejo. La inicialización de las estructuras se realiza como si se tratara de un tipo normal.

Una vez que una estructura ha sido definida, podemos declarar variables del tipo de la estructura usando el nombre anteponiendole o no *struct.* En la línea **010** se ha definido la variable *a* de tipo *complex* anteponiendo la palabra *struct*; esto no es estrictamente necesario, bastaría con *complex a*.

Para acceder a los miembros de una estructura se utiliza el operador ., o sea, un punto.
En las líneas de código **011** y **012** se accede al miembro *real* y al miembro *imag* de la variable *a* de tipo *complex* y se le asigna en ambos caso el valor *1*.

Una propiedad que se deriva de la definición secuencial de estructura es que la longitud de una estructura es la suma de las longitudes de los tipos que la componen.

5 Ejercicios

5.1 Definir una estructura para puntos en el espacio. Definir funciones que calculan distancias entre puntos con argumentos de estructuras.

Solución al ejercicio.

```
// Ficha 8
/* Definir una estructura para puntos en el espacio*/

#include <iostream.h>
#include <math.h>

//estructura para definir un punto en el espacio
struct puntesp
{
double x,y,z;// coordenadas de los puntos
};

//declaración de la función distancia entre dos puntos
double distancia (puntesp Pi, puntesp Pj);
```

```
//programa principal

int main (void)
{
puntesp P1,P2;
cout << "Introduzca las coordenadas del primer punto:"<<endl;
cin >> P1.x, P1.y, P1.z;
cout << "Introduzca las coordenadas del segundo punto:"<<endl;
cin >> P2.x, P2.y, P2.z;

cout << "la distancia entre los dos puntos es="<<distancia (P1,P2)<<endl;
return 0;
}

//definición de la función distancia
double distancia (puntesp Pi, puntesp Pj)
{
return sqrt((((Pj.x-Pi.x)*(Pj.x-Pi.x))+((Pj.y-Pi.y)*(Pj.y-Pi.y))+
((Pj.z-Pi.z)*(Pj.z-Pi.z))));
}
```

Como comentario, obsérvese que la función *distancia* toma como argumentos variables del tipo *punto* en el espacio y se olvida de las coordenadas. Las coordenadas son datos que pertenecen al concepto abstracto de punto, aunque en última instancia son las que se utilizan para calcular la distancia. Pero en el pensamiento, en ese acercamiento de la programación al lenguaje natural, cuando hablamos de distancia entre puntos estamos hablando del concepto: *distancia entre puntos y no entre coordenadas*. La idea parece sutil y poco importante, pero en programas complejos puede ser determinante en la calidad del código.

En la función que calcula la distancia se están pasando las estructuras enteras, por lo tanto se produce un despilfarro de memoria y de tiempo al tener que duplicar la información local. Por ello sería mejor pasar la información de los argumentos puntos utilizando punteros. Esto se verá con más detalle en la ficha 12.

5.2 Definir estructuras para cuadriláteros y triángulos. Definir funciones que calculan áreas.

Solución al ejercicio

```
//Ficha 8
/* Definir estructuras para cuadrados y triángulos*/

#include <iostream.h>

//estructura cuadrados
struct cuadrado
{
double lado;
};

//estructura triangulo
struct triangulo
{
double  base, altura;
};

//declaración funciones area1 y area2
double area1 (cuadrado p);

double area2 (triangulo p);
```

```
//programa principal
int main (void)
{
int x;
cout<<"Indique la figura que quiere calcular,
1(cuadrado),2(triángulo):"<<endl;
cin>>x;

if (x==1)
{
  cuadrado a;
  cout<<"Introduzca el valor del lado:"<<endl;
  cin>>a.lado;
  cout<<"El valor del área es:"<<area1(a)<<endl;
}
else
{
  triangulo b;
  cout<<"Introduzca el valor de la base:"<<endl;
  cin>>b.base;
  cout<<"Introduzca el valor de la altura:"<<endl;
  cin>>b.altura;
  cout<<"El valor del área es:"<<area2(b)<<endl;
}
return 0 ;
}

//definición función area1

double area1 (cuadrado p)
{return (p.lado*p.lado);}

//definición función area2

double area2(triangulo p)
{return (p.base*p.altura)/2;}
```

5.3 Definir una estructura para números complejos y sus operaciones.

Solución al ejercicio.

```
//Ficha 8
/* Definir una estructura para números complejos y sus operaciones

#include <iostream.h>

//estructura complejo
struct complex
{
double real, imag;
};

//declaración función suma
complex suma_complex (complex a, complex b);

//programa principal
int main (void)
{
complex c1, c2, c;

cout<<"Introduzca parte real e imaginaria del primer número:"<<endl;
```

```
cin>>c1.real, c1.imag;
cout<<"Introduzca parte real e imaginaria del segundo número:"<<endl;
cin>>c2.real, c2.imag;

c= suma_complex (c1,c2);
cout<<"suma compleja="<<c.real>>"+"<<c.imag>>"i""<<endl;
return 0;
}

//definición de la función suma
complex suma_complex (complex a, complex b)
{
complex c;
c.real=a.real+b.real;
c.imag=a.imag+b.imag;
return c;
}
```

6 Ejemplo

Se define el concepto matriz mediante una estructura, se agrupan los datos en un único bloque abstracto:

```
#include <iostream.h>
#include <fstream.h>

#define MAX_SIZE 25

struct Matrix { int size; int s[MAX_SIZE]; };

void report_error(void)
{
 ofstream MyReport("resultado.txt");
 cerr << "Error, tamaño demasiado grande" << endl;
 MyReport << " *********************************** inicio"
          << endl;
 MyReport << " ERROR: dimension del vector debe ser menor de "
          << MAX_SIZE   << endl;
 MyReport << " ************************************** fin"
         << endl;
 MyReport.close();
 return;
}

int report_final (struct Matrix mat)
{
 ofstream MyReport("resultado.txt");
 MyReport << " *********************************** inicio" << endl;
 MyReport << " Vector de " << mat.size << " componentes" << endl;
 for (int i=0; i<mat.size; i++) MyReport << mat.s[i] << " ";
 MyReport << endl;
 MyReport << " ************************************** fin" << endl;
 MyReport.close();
 return 1;
}

int main (void)
{
 struct Matrix MyMatrix;

 cout << "Introduzca la dimensión del vector"<<endl;
 cin >> MyMatrix.size;
```

```
 if (MyMatrix.size > MAX_SIZE) report_error();

 for (int i=0; i<MyMatrix.size; i++) MyMatrix.s[i]=0;
                                  // inicializando a cero cada componente
 if (report_final(MyMatrix))
      cout << "Vamos a trabajar con un vector de " << MyMatrix.size
              << " componentes" <<endl;
 else cerr << "error general" << endl;

return 0;
}
```

En este caso se define el concepto matriz mediante una estructura, se agrupan los datos en un único bloque abstracto y se explicita una operación tipo suma:

```
#include <iostream.h>
#include <fstream.h>

#define MAX_SIZE 25

struct Matrix { int size; int s[MAX_SIZE]; };

struct Matrix Suma_de_Matrices(struct Matrix Iz, struct Matrix Dr);
void print_matrix (struct Matrix mat);

int main (void)
{
 int i;
 struct Matrix A; A.size=5;
 for (i=0; i<A.size; i++) A.s[i]=1;
 struct Matrix B; B.size=5;
 for (i=0; i<B.size; i++) B.s[i]=1;

 struct Matrix C=Suma_de_Matrices(A,B);
 print_matrix(C);

return 0;
}

struct Matrix Suma_de_Matrices(struct Matrix Iz, struct Matrix Dr)
{
 struct Matrix temp; temp.size=Iz.size;
 for (int i=0; i<temp.size; i++) temp.s[i]=Iz.s[i]+Dr.s[i];
 return temp;
}


void print_matrix(struct Matrix mat)
{
 ofstream MyReport("resultado.txt");
 MyReport << " *********************************** inicio" << endl;
 MyReport << " Vector de " << mat.size << " componentes" << endl;
 for (int i=0; i<mat.size; i++) MyReport << mat.s[i] << " ";
 MyReport << endl;
 MyReport << " ************************************** fin" << endl;
 MyReport.close();
 return;
}
```

7 Ampliación de conceptos

7.1 Uniones

Como ya se vio el C++ permite definir tipos más complejos que los datos básicos. Para ello agrupa un conjunto de variables del mismo o diferente tipo en un nuevo tipo. Una *unión* es similar a una estructura en sintaxis y declaración, excepto que las variables que componen una unión comparten el mismo espacio físico, o sea , la misma dirección de memoria.

Ejemplo:

```
union  FloInt
{
int i;
float f;
} FloatEntero;

FloatEntero.i = 10;
FloatEntero.f= 43.3;      //tanto i como f compartiran la misma memoria
int a = FloatEntero.i;    //a depende de la codificación de f
```

Se pueden combinar estructuras y uniones para formar grandes tipos; sin embargo, en la orientación a objetos no se usan demasiado.

7.2 Campos de Bits

Cuando se necesitan compactar muchos datos en poco espacio, una buena solución es utilizar la unión. Una solución más drástica es bajar el nivel de bit y utilizar los *campos de bits,* lo cual es muy útil para acceder a controladores, dispositivos, etc. Uno de los problemas es que algunos compiladores asignan los bits de izquierda a derecha y otros al revés, con lo que un programa que use campos de bits no será independiente de la maquina.

7.3 Typedef

Typedef permite renombrar los tipos, por ejemplo:

```
typedef  int  entero;
```

A partir de esta declaración se puede usar la palabra *entero* como si se tratase de un *int*. A diferencia de otros lenguajes, el tipo declarado será totalmente compatible con el tipo con el que se declara, se podrán intercambiar variables, valores, etc., sin ningún problema.

```
int a = 10;
entero e = 20;
e = a;
a = e ;
```

La utilidad del uso de *typedef* estriba en que al crear nuevos tipos se puede independizar el código del compilador-procesador. Así, si se cambia a una nueva máquina sólo se deberá cambiar el *typedef* apropiado por su nuevo tipo en el encabezado del código, sin tocar el resto.

Ficha 9a : Clases. Parte I

1 Introducción

En la ficha anterior se ha visto el concepto de abstracción de datos. Su utilización permite concebir programas comprensibles y con mayor facilidad de mantenimiento. Sin embargo, en un programa hay que operar con los datos, y hasta ahora las funciones que operan con los datos se han definido externas a dichos datos. Por ejemplo, en la ficha 8, se define la estructura número complejo y se define una función suma de complejos que utiliza dos argumentos y devuelve un número complejo.

Dado que algunos datos tienen asociadas operaciones de forma natural, suma de complejos, productos, etc. ¿no sería más lógico pensar que los datos y sus operaciones deben ir juntos? En este caso la abstracción no se limita a la organización de la información, sino que también incluye el concepto de lo que se puede hacer con ella.

En este contexto surge el concepto de *clase* (*class*) como abstracción que reúne datos y operaciones con los datos.

2 Objetivos generales

- Conocer la filosofía de las clases = datos + operaciones.
- Encapsulamiento public. Constructores y destructores.

3 Código de trabajo

```
//Ficha 9a
/*Clases. datos + operaciones, encapsulamiento public, constructores y destructores*/
#include <iostream.h>
class complex
{
public:
double real, imag;
complex (double a, double b) {real=a; imag=b;}
~ complex () {}
};
int main (void)
{
complex a (1.,1.);
cout <<"complex="<<a.real<<"+"<<a.imag<<"i"<<endl;
return 0;
}
```

4 Conceptos

4.1 Qué son las clases

Se define una *clase* como un conjunto de datos (*atributos*) y operaciones (*métodos*).
Todos los elementos definidos dentro de una clase se llaman *miembros* (*atributos y métodos*).

Los *atributos* de una clase son variables que pueden ser tipos básico del lenguaje, punteros, tipos definidos por el usuario, etc. Los *métodos* son funciones que operan con los datos de la clase y con datos externos que se pasan como argumentos de la función.

Las clases en C++ tienen las siguientes características:
1. Un nombre identificador (línea **004**; *complex*).
2. Un conjunto de atributos (datos miembro) (línea **007**; *real* y *imag,* ambos de tipo *double*).
3. Un conjunto de métodos (funciones miembro)(línea **008**; *complex (double a, double b)).*
4. Unos niveles de acceso para proteger ciertas partes de la clase (línea **006**; *public*).

Un *objeto* es una entidad que tiene un estado definido por los valores que toman los atributos y un conjunto definido de operaciones, los métodos que operan sobre este estado. La creación de un objeto se llama *instanciación* de la clase. Un objeto es por tanto el valor que toma una clase; sería comparable a los conceptos de *int* clase y *a* objeto:

```
int a;
```

La clase será el concepto teórico y el objeto su utilización práctica. Por abuso del lenguaje clase y objeto suelen solaparse.

La instanciación de la clase *complex* se realiza en la línea **013**, donde se crea un objeto *a* que toma como valores real e imaginario el 1. Una vez creado el objeto, se puede acceder a sus miembros a través del operador de acceso **.** (un punto), que ya se había visto en la ficha 8 (línea **014**, con *a* **.** *real* accedemos al valor de la variable *real* del objeto *a* de la clase *complex*).

4.2 Constructores y Destructores

Un *constructor* en C++ es una operación, una función, que inicializa los atributos del objeto en el momento de su creación. Cuando se crea un objeto surge la necesidad de asignar valores iniciales a las variables que contiene, esa es la misión del constructor. Salvando las diferencias conceptuales el constructor de los datos *int* sería:

```
int a=1;
```

Donde *int* sería la clase, *a* el objeto el signo = el constructor que coloca el valor 1 en *a*.

Para definir un costructor se utiliza el propio nombre de la clase, como en la línea **008**; *complex (double a, double b*). Se pueden definir varios constructores para una misma clase, todos ellos tendrán el mismo nombre (el de la clase) pero, en general, se distinguirán por sus argumentos. Si no se define ningún constructor, en general el compilador creará uno por defecto *complex ()* { }, un constructor vacío que no hace nada.

En la línea **013** se define un objeto *complex a*, el programa llama al constructor de la clase que se ha definido en la línea **008** *complex (double a, double b*) y asigna los valores de los argumentos a los atributos del objeto. Por lo tanto, en la línea **013** se crea el objeto *a* y se inicializan sus atributos *a.real* y *a.imag* con los valores 1.

Los *destructores* sirven para que el objeto libere memoria y termine con asuntos pendientes como cerrar ficheros, etc. antes de ser destruido. El destructor se llama automáticamente justo antes de que el objeto que lo contiene sea destruido. Tal y como sucedía con el constructor, si no se define un destructor el compilador creará uno por defecto.

Se define el destructor con el nombre de la clase precedido por el carácter ˜. El destructor no puede tener parámetros (argumentos) ni retornar valores (ni siquiera *void*), es decir, el destructor es único. En el ejemplo, en la línea **009** ˜*complex* (){} se define el destructor como una función que en este caso no realiza ninguna acción.

5 Ejercicios

5.1 Definir complejos en notación polar. Añadir constructores para dicha notación.

Solución al ejercicio.

```
// Ficha 9a
/* Definir complejos en notación polar.
Añadir constructores para dicha notación. */

#include <iostream.h>
#include <math.h>
#define PI 3.14159

class complexPolar
{
 public:
  double mod, arg;

  complexPolar (double a, double b)
  {
    mod = sqrt ((a*a)+(b*b));
    arg = 180.*atan(b/a)/PI;
  }

  ~complexPolar () {}

};

int main (void)
{
double a, b;

cout<< "introduzca la parte real del punto:"<<endl;
cin>> a;
cout<< "introduzca la parte imaginaria del punto:"<<endl;
cin>> b;

complexPolar  P (a, b);

cout   <<   "la   notacion   polar   del   numero   complejo   introducido   es:"<<
"("<<P.mod<<","<<P.arg<<")"<<endl;

return 0;
}
```

5.2 Clases para puntos, segmentos, cuadriláteros y triángulos.

Solución al ejercicio.

```
// Ficha 9a
/* Clases para puntos, segmentos, cuadriláteros y triángulos*/

#include <iostream.h>

//clase punto
class Point
{
public:
int x,y; //datos tipo entero

Point (void){} //constructor sin argumento y vacio (punto origen)
Point (int a, int b) {x=a; y=b;}//constr.con dos argumentos enteros
Point (int a) { x=a; y=a;}//constr.con un argum.(punto diagonal)

˜Point (){} //Destructor
};

//clase segmento
class Segment
{
public:

Point P1,P2; // datos tipo punto

Segment (void) {} // constructor sin argumento y vacio
Segment (Point A1, Point A2){P1=A1; P2=A2;}//constr. dos argumentos

˜Segment //destructor
};

//clase cuadrado
class Quad
{
public:

Segment S1,S2; // datos tipo segmento

Quad (Segment A1, Segment A2){S1=A1; S2=A2;}//constr. dos argum.

˜Quad //destructor
};

//programa principal
int main (void)
{
Point Ori (0); //objetos de la clase punto. Punto origen (0,0)
Point P1  (1);//Punto (1,1)
Point P2  (2,5);//Punto (2,5)

Segment S1 (Ori, P1);//objetos de la clase segmento
Segment S2 (P1, P2);

Quad C (S1,S2);//objeto de la clase cuadrado

return 0;
}
```

6 Ejemplo

Se define el concepto matriz mediante una clase, se agrupan los datos y las operaciones en un único bloque abstracto:

```
#include <iostream.h>
#define MAX_SIZE 25

class Matrix
{
 public:
       // ***************************** datos
       int size; double s[MAX_SIZE];

       // ***************************** operaciones
       // constructores
       Matrix () : size(0) {}
       Matrix (int m)
       {
        if (m>MAX_SIZE)
         {cerr << "la dimension maxima es "
                << MAX_SIZE << endl; m=MAX_SIZE;}
           size=m;  for (int i=0; i<size; i++) s[i]=0.;
           for (i=0; i<size; i++)
              {cout << endl << "coef (" << i << ")="; cin >> s[i];}
        }

       ~Matrix() { size=0; }
       void Suma_de_Matrices (Matrix B);
};

// funcion suma de matrices
void Matrix :: Suma_de_Matrices (Matrix A)
{
  if (A.size != size)
   { cerr << "suma con dimensiones diferentes. Error" << endl; return;}
  for (int i=0; i<size; i++)
        cout << "suma(" << i << ")=" << s[i] + A.s[i] << endl;
  return;
}

// el programa ejecutable
int main()
{
 Matrix A(3); Matrix B(3);
 A.Suma_de_Matrices (B);
 return 0;
}
```

Ficha 9b : Clases. Parte II

1 Objetivos generales

- Encapsulamiento *private*, constructor copia y redefinición de operadores.

2 Código de trabajo

```
//Ficha 9b
/*Encapsulamiento private, constructor copia y redifinición de operadores*/
#include <iostream.h>
class complex
{
private:
double real, imag;
public:
complex (double a, double b)            //constructor
{real=a ; imag =b;}
~complex () {}       //destructor
{}
complex (complex& a)                    //constructor copia
{real=a.get_real(); imag=a.get_imag();}
complex&  operator = (complex& m)       //asignación
{real=m.get_real(); imag=m.get_imag(); return *this;}
double get_real (void) {return real;} //acceso al encapsulamiento
double get_imag (void) {return imag;}
friend ostream& operator << (ostream& os, complex& a)
{      //redefinición del operador <<
os << "Es un Complex=";
os << a.get_real() <<"+"<< a.get_imag () <<"i"<<endl;
return os;
}
};

int main (void) //programa principal
{
complex a(1.,1.); //definimos un objeto a complejo
complex b; //definimos un objeto b complejo
b=a; //asignamos un complejo a otro
complex c(a) ; //copiamos un complejo de otro
cout <<"complex a ="<<a.get_real()<<"+"<<a.get_imag()<<"i"<<endl;
cout <<"complex b ="<<b.get_real()<<"+"<<b.get_imag()<<"i""<<endl;
cout << c << endl;
return 0;
}
```

3 Conceptos

3.1 Encapsulación

Se define la *encapsulación* como la habilidad de poner todos los datos y operaciones sobre los datos en una estructura controlada y abstracta, de forma que quede aislada del resto del sistema. Las clases permiten al programador la libertad de encapsular variables y métodos en una única estructura, de forma que los únicos que podrán acceder a estos datos y operaciones son los objetos de esa clase. Con esto se consigue la encapsulación, lo que implica seguridad e independencia.

Se define *campo o ámbito* a la parte del programa donde es accesible un identificador, variable, función, etc. Dicho concepto aplicado a todo el programa se vio con detalle en la ficha 7 y puede restringirse a los miembros de la clase. Por lo tanto, el *campo de clase* extiende la definición de ámbito a los miembros de una clase; en consecuencia, al mismo tiempo que permite el acceso a otros miembros de la misma clase, impide el acceso a los miembros de otras clases. De esta forma se consigue la encapsulación; cualquier error o modificación de una clase no interfiere en las demás.

Los *niveles de acceso* a los miembros de una clase se clasifican en:

1. *Public*: público, accesibles por métodos de cualquier clase. Línea **008**.
2. *Protected*: protegido, accesible sólo por métodos de su clase y descendientes.
3. *Private*: privado, sólo accesible por métodos de su clase. Línea **006**.

Su sintaxis es:

EspecificadorAcceso : (**006** y **008**)
ListaMiembros (**007 y 009-025**)

Se debe destacar, que por defecto, el acceso a los miembros de una clase es privado, es decir, no son accesibles desde el exterior de la clase.

Los miembros privados de una clase están encapsulados y su modificación y uso se realiza a través de los métodos de la propia clase. En definitiva, si se desea modificar el valor de las variables *private* se deben definir métodos de acceso para operar con ellas. En el código de trabajo anterior se han definido dos métodos *get_real* y *get_imag*¸ líneas **017** y **018,** para acceder a los atributos privados de la clase *complex*. La llamada a estos métodos se realiza mediante el operador de acceso **.** como si se tratara de una variable en una estructura, línea **032**, *a.get_real( )* (aceso a la parte *real* del objeto *a*). Nótese que la privacidad impide escribir algo como *a.real* en esa misma línea.

3.2 Constructor copia

Es un constructor que tiene un único parámetro, una referencia (constante o no) al mismo tipo que la clase donde se define el constructor. Su finalidad es copiar los datos de un objeto a otro (líneas **013** y **014**). Las llamadas al constructor copia se realizan como se indica en la línea **031.**

En el caso de que no defina un constructor copia, el compilador deberá crear uno por omisión y copiar todos los atributos del objeto que se pasa por referencia al objeto actual.

3.3 Operador de asignación (Método operador =)

El operador de asignación realiza una copia de todos los atributos del objeto fuente. Su parámetro es una referencia a la misma clase. Es un método que suele venir implementado por omisión, pero también se puede definir tal y como se hace en las líneas **015** y **016**.

La llamada al operador de asignación se realiza de la forma indicada en la línea de código **030** (b=a;) siguiendo el sentido lógico y natural de nuestro pensamiento. Si se hubieran englobado las líneas de código **029** y **030** en una sola (complex b=a;), se estaría llamando al constructor copia y no al de asignación, ya que esta última sentencia es equivalente a (complex b(a);). La diferencia entre el constructor copia y el operador de asignación es que el primero crea un nuevo objeto y el segundo no, trabaja sobre objetos ya creados.

Como norma general se deberá definir un constructor copia y un operador de asignación para todas las clases, así como un constructor por defecto y un destructor. Aunque no hagan nada.

3.4 Parámetro implícito this

En C++ *this* es un puntero al propio objeto que llama en el método de la línea **016;** su uso y concepto se verá en la ficha 12c.

4 Ejercicios

4.1 Definir las operaciones con complejos.

Una posible solución al ejercicio.

```
// Ficha 9b
#include <iostream.h>

class Complex
{
 public :
 double real,imag ;

Complex (double a, double b)
  { real=a ; imag =b ; }
 ~Complex ()  {}

Complex (Complex& a)
{ real = a.real ; imag=a.imag;}

Complex& operator = (Complex& m)
{real = m.real; imag = m.imag ; return *this;}

friend ostream& operator << (ostream& os, Complex& a)
{
os << "COMPLEX=";
os << a.real<<"+"<<a.imag<<"i"<<endl;
return os;
}

Complex suma_compleja (Complex& a, Complex& b)
{
Complex c (0.,0.);
c.real = a.real + b.real;
```

```
c.imag = a.imag + b.imag;
return c;
}

friend Complex& operator +(Complex& a, Complex& b)
{
Complex c (0.,0.);
c.real = a.real + b.real;
c.imag = a.imag + b.imag;
return c;
}

friend Complex& operator -(Complex& a, Complex& b)
{
Complex c (0.,0.) ;
c.real = a.real - b.real;
c.imag = a.imag - b.imag;
return c;
}

};

int main (void)
{
Complex a (1., 2.); Complex b (0.,6.);
Complex c (a) ; Complex d=b;
Complex e = a + b;
Complex f= c - d;
cout<<c<<endl; cout<<d<<endl; cout<<e<<endl; cout<<f<<endl;
e=e.suma_compleja(a,b);
cout << " suma como función " << e <<endl;
return 0;
}
```

4.2 Definir distintas operaciones con las clase de poligonos: cálculo de áreas, comparación de figuras. Definir operaciones con los segmento: distancias.

Solución al ejercicio.

```
//Ficha 9b
/* Clases para puntos, segmentos y cuadriláteros */

#include <iostream.h>
#include <math.h>

class Point //clase punto
{
public:
int x,y; //datos tipo entero

Point (void){} //constructor sin argumento y vacio (punto origen)
Point (int a, int b) {x=a; y=b;}//constr.con dos argumentos enteros
Point (int a) { x=a; y=a;}//constr.con un argum.(punto diagonal)

~Point (){} //Destructor

Point&  operator= (Point& P) //operador asignación
 { x = P.x ; y = P.y ;
   return *this;} // puntero implícito this
 };
```

```
class Segment //clase segmento
{
public:

Point P1,P2; // datos tipo punto

Segment (void) {} // constructor sin argumento y vacio
Segment (Point A1, Point A2){P1=A1; P2=A2;}//constr. dos argumentos

˜Segment //destructor

//definición función de cálculo de la distancia entre dos puntos
double longitud ()  {return sqrt(((P2.x-P1.x)*(P2.x-P1.x))+
                              ((P2.y-P1-y)*(P2.y-P1.y)))};}

Segment&  operator =  (Segment& S) //operador asignación
 {P1=S.P1; P2=S.P2;
   return *this;}
};

class Quad //clase cuadrado
{
public:

Segment S1,S2; // datos tipo segmento

Quad (Segment A1, Segment A2){S1=A1; S2=A2;}//constr. dos argum.

˜Quad //destructor
//definición función de cálculo área dos segmentos (un cuadrado)
double Area ()  {return ((S1.longitud())*(S2.longitud()));}
};

//programa principal

int main (void)
{
Point Ori (0,0); Point P1 (2,0); Point P2  (2,5);
Segment S1 (Ori, P1); Segment S2 (P1, P2);
Quad C (S1,S2);
cout <<"la longitud de la base es:"<< S1.longitud()<<endl;
cout <<"la longitud de la altura es:"<< S2.longitud()<<endl;
cout <<"el área del cuadrado es:"<< C.Area()<<endl;
return 0;}
```

5 Ejemplos

5.1 Versión 1. Se define una clase para matrices y se introducen los operadores y constructores fundamentales: copia, comparación e igualdad

```
#include <iostream.h>
#include <stdlib.h>
#define MAX_SIZE 25

class Matrix
{
 public:
int size; double s[MAX_SIZE];

Matrix () : size(0) {}
```

```
Matrix (int m)
{
 if (m>MAX_SIZE)
  {cerr << "la dimension maxima es " << MAX_SIZE << endl; m=MAX_SIZE;}
          size=m;  for (int i=0; i<size; i++) s[i]=0.;
 }

 void InicializarMatrix (void);

 Matrix (Matrix& m);
 Matrix operator = (Matrix m);


 ~Matrix() { size=0; }
 Matrix Suma_de_Matrices (Matrix B);

};


int main()
{
 Matrix A(3); A.InicializarMatrix();

 Matrix B(A); Matrix C=A;

 int i;
 cout << "matriz A = ";
 for (i=0; i<A.size; i++) cout << A.s[i] << " ";  cout << endl;
 cout << "matriz B = ";
 for (i=0; i<B.size; i++) cout << B.s[i] << " ";  cout << endl;
 cout << "matriz C = ";
 for (i=0; i<C.size; i++) cout << C.s[i] << " ";  cout << endl;

 Matrix D=A.Suma_de_Matrices (B);
 cout << "matriz D = ";
 for (i=0; i<D.size; i++) cout << D.s[i] << " ";  cout << endl;

 return 0;
}


Matrix::Matrix(Matrix& m)
{
 size=m.size;
 for (int i=0; i<size; i++) s[i]=m.s[i];
}


Matrix Matrix::operator = (Matrix m)
{
  Matrix Temp(m.size);
  for (int i=0; i<size; i++) Temp.s[i]=m.s[i];
  return Temp;
}

Matrix Matrix :: Suma_de_Matrices (Matrix A)
{
  if (A.size != size)
   { cerr << "suma con dimensiones diferentes. Error" << endl; exit(1);}
  Matrix C(A.size);
  for (int i=0; i<size; i++) C.s[i]=s[i] + A.s[i];
  return C;
}

void Matrix:: InicializarMatrix (void)
```

```
{
 for (int i=0; i<size; i++)
       { cout << "coeficiente (" << i <<")="; cin >> s[i]; }
 return;
}
```

5.2 Versión 2. En este caso se hace lo mismo que el anterior pero jugando con el concepto del operador suma. La intención es acercar el lenguaje natural a las facilidades sintácticas. Por lo tanto la operación suma ya no se concibe como una función sino como una redefinición del operador +.

```
#include <iostream.h>
#include <stdlib.h>
#define MAX_SIZE 25

class Matrix
{
 public:
  int size; double s[MAX_SIZE];

  Matrix () : size(0) {}
  Matrix (int m)
  {
   if (m>MAX_SIZE)
   {cerr << "la dimension maxima es " << MAX_SIZE << endl; m=MAX_SIZE;}
          size=m;  for (int i=0; i<size; i++) s[i]=0.;
  }

 void InicializarMatrix (void);

 Matrix (Matrix& m);
 Matrix operator = (Matrix m);


 ~Matrix() { size=0; }

  Matrix operator + (Matrix m);
};


int main()
{
 Matrix A(3); A.InicializarMatrix(); Matrix B(3); B.InicializarMatrix();

 Matrix C=A+B;

 int i;
 cout << "matriz A = ";  for (i=0; i<A.size; i++) cout << A.s[i] << " ";  cout <<
endl;
 cout << "matriz B = ";  for (i=0; i<B.size; i++) cout << B.s[i] << " ";  cout <<
endl;
 cout << "matriz C = ";  for (i=0; i<C.size; i++) cout << C.s[i] << " ";  cout <<
endl;
  return 0;
}
Matrix::Matrix(Matrix& m)
{
 size=m.size;
 for (int i=0; i<size; i++) s[i]=m.s[i];
}
```

```
Matrix Matrix::operator = (Matrix m)
{
  Matrix Temp(m.size);
  for (int i=0; i<size; i++) Temp.s[i]=m.s[i];
  return Temp;
}

void Matrix:: InicializarMatrix (void)
{
 for (int i=0; i<size; i++)
       { cout << "coeficiente (" << i <<")="; cin >> s[i]; }
 return;
}


Matrix Matrix::operator + (Matrix m)
{
 if (size!=m.size)
    { cerr << "suma con dimensiones diferentes. Error" << endl; exit(1);}

 Matrix Temp (m.size);
 for (int i = 0; i < size; i++)  Temp.s[i] = s[i] + m.s[i];
 return Temp;
}
```

5.3 Versión 3. En este caso se hace lo mismo que el anterior, pero jugando con los conceptos de público y privado.

```
#include <iostream.h>
#include <stdlib.h>
#define MAX_SIZE 25

class Matrix
{
private:
  int size; double s[MAX_SIZE];

public:
  Matrix () : size(0) {}
  Matrix (int m)
  {
   if (m>MAX_SIZE)
   {cerr << "la dimension maxima es " << MAX_SIZE << endl; m=MAX_SIZE;}
          size=m;  for (int i=0; i<size; i++) s[i]=0.;
  }

  int get_size(void) {return size;}

  void InicializarMatrix (void);

 Matrix (Matrix& m);
 Matrix operator = (Matrix m);


 ~Matrix() { size=0; }

  Matrix operator + (Matrix m);
  friend ostream& operator << (ostream& os, Matrix& m);
};

int main()
{
 Matrix A(3); A.InicializarMatrix(); Matrix B(3); B.InicializarMatrix();
```

```
 Matrix C=A+B;

 cout << "matriz A = " << A << endl;
 cout << "matriz B = " << B << endl;
 cout << "matriz C = " << C << endl;

 // cout << A.size; error de compilacion
 cout << A.get_size();
 return 0;
}


Matrix::Matrix(Matrix& m)
{
 size=m.size;
 for (int i=0; i<size; i++) s[i]=m.s[i];
}


Matrix Matrix::operator = (Matrix m)
{
  Matrix Temp(m.size);
  for (int i=0; i<size; i++) Temp.s[i]=m.s[i];
  return Temp;
}

void Matrix:: InicializarMatrix (void)
{
 for (int i=0; i<size; i++)
       { cout << "coeficiente (" << i <<")="; cin >> s[i]; }
 return;
}


Matrix Matrix::operator + (Matrix m)
{
 if (size!=m.size)
    { cerr << "suma con dimensiones diferentes. Error" << endl; exit(1);}

 Matrix Temp (m.size);
 for (int i = 0; i < size; i++)  Temp.s[i] = s[i] + m.s[i];
 return Temp;
}

ostream& operator << (ostream& os, Matrix& m)
{
 os << "Matrix ++++ "; if (!m.size) os<<"void";
 for (int i=0; i<m.size; i++) os << m.s[i] <<" " ;
 os << "\n";
 return os;
}
```

6 Ampliación de conceptos

6.1 Operador de campo ::

Se ha visto que los métodos se definen de la misma forma que las funciones normales y hasta ahora se han mantenido dentro de las llaves que limitan la clase (*métodos internos*). Cuando las funciones sean muchas y muy grandes, es conveniente dejar unicamente los prototipos dentro del cuerpo de la clase y definir las funciones fuera (*métodos externos*).

Para definir una función fuera de la clase donde ha sido declarada es necesario utilizar el *operador de campo* **::**. Ejemplo:

```
class  NombreClase
{
   ....
   //declaración del  Metodo1
   tipo0    Metodo1  (tipo1  a ,  tipo2   b)
    ...
};

//programa principal
  int main (void)
{
...
  };

// definición del Método1 fuera de la clase en la que ha sido declarado
tipo0    NombreClase  ::  Metodo1 (tipo1  a , tipo2  b )
{.........}
```

Los método internos (definidos dentro del cuerpo de la clase) son funciones que no se llaman, sino que su código se sustituye cada vez que son llamadas (funciones *inline*). Por contra, los métodos externos se comportan como funciones normales que sí son llamadas, por ello los métodos *inline* son más rápidos en tiempo de ejecución. Normalmente, las funciones pequeñas se declaran y definen dentro del cuerpo de la clase, mientras que las funciones grandes se declaran en la clase pero se definen fuera.

6.2 Funciones amigas

Dentro de una clase podemos definir una función *amiga (friend)* que permita compartir los datos o funciones con otras clases, es decir que permita acceder a sus miembros privados. Lo interesante de este tipo de declaración es que la clase es quien establece sus amigos y limita aquellos que pueden acceder a sus miembros privados. De este modo ninguna función se puede autodefinir como amiga y acceder a la privacidad de una clase sin tener conocimiento de ella dentro de la propia clase. Con esto aseguramos que continue la seguridad y protección que proporcionan las clases.

En el código de trabajo aparece en la línea **019**:

```
friend ostream& operator << (ostream& os, complex& a)          //redefinición operador <<
                                                               // función amiga
```

Esta función permite utilizar el operador << de escritura en *stream* para escribir un formato de salida de datos de la propia clase. Nótese que las funciones amigas no pertenecen a ninguna clase, por lo tanto, si una función declarada en una clase se define fuera de ella, no necesita el operador de campo :: y su llamada tampoco necesita hacerse a través de un objeto de la clase. En la línea **034** se escribe cout<<c y no cout<<c.os o cosas parecidas.

6.3 Conversiones

Cuando se crean clases, que no son más que tipos complicados, no se tiene definida ninguna forma de conversión de una clase a otra. El C++ permite escribir estas conversiones y la forma como deben

hacerse. Como se ha visto antes en la ficha 2, los tipos fundamentales del C++, *int*, *char*, *float*, etc., tienen dos tipos de conversiones. Las implícitas,que vienen definidas por defecto en el lenguaje, y las explícitas ,que escribe el programador mediante los *castings* para forzar la conversión y evitar las posibles malas interpretaciones.

Las clases definidas por el usuario pueden aceptar conversiones implícitas y forzar explícitas. Para ello, se tienen que programar las instrucciones decodificadoras en unos *constructores de conversión,* o bien, unos *operadores de conversión*, que permiten pasar de cualquier tipo a nuestra clase o viceversa, de la propia clase a cualquier tipo.

6.4 Objetos y miembros constantes

Se permite definir objetos constantes, mediante el identificador *const*, del mismo modo que se podían crear variables constantes con los tipos por defecto. En este caso, las únicas operaciones que se pueden hacer con objetos constantes es construirlos y destruirlos.

Se suelen definir como constantes aquellos métodos que no modifican el objeto, por ello los constructores y el destructor no pueden ser nunca definidos como tal. El identificador *const* hay que incluirlo tanto en la declaración como en la definición de la función.

Además de métodos constantes se pueden definir atributos constantes en cuyo caso no se pueden modificar una vez inicializados.

Ficha 9c: Ejemplo de recapitulación sobre la abstracción datos

1 Ejercicio: Calcular la distancia entre dos puntos

1.1 Solución al ejercicio sin abstracción de datos

```
// ficha 9c
/* sin abstracción de datos.*/

#include <iostream.h>
#include <math.h> // contiene las funciones matemáticas standar

void main (void)
{
//declaración de 4 números enteros

int x1,x2;
int y1,y2;

//entrada por pantalla de las coordenadas x e y de dos puntos

cout << "Introduzca las coordenadas del punto 1:"<<endl;
cin >> x1, y1;
cout << "Introduzca las coordenadas del punto 2:"<<endl;
cin >> x2, y2;

//función para cálculo de la distancia de dos puntos

double dist = (x2-x1)*(x2-x1)+(y2-y1)*(y2-y1);

//escritura por pantalla

cout << "La distancia es:"<<sqrt(dist)<<endl;
}
```

1.2 Solución al ejercicio utilizando abstracción de datos con estructuras

```
//Ficha 9c
/* con abstracción de datos (estructuras)*/

#include <iostream.h>
#include <math.h>
```

```
//estructura punto
struct Point //estructura punto
{
int x,y; //datos tipo entero
};
//declaracion de funciones de lectura datos y calculo distancia
struct Point leer_datos (void);
double calc_dist (struct Point p1, struct Point P2);

//programa principal
void main (void)
{

//declaración de dos puntos
struct Point P1;
struct Point P2;

//inicialización de los puntos
P1=leer_datos();
P2=leer_datos();

//cálculo de distancia entre dos puntos y escritura en pantalla
double dist = calc_dist (P1,P2);
cout << "La distancia es:"<<dist;
}

//definición de la funcion de lectura
struct Point leer_datos ()
 {
 struct Point P;
 cin>>P.x;
 cin>>P.y;
 return P;
 }

//definición de la función de cálculo de distancia
double calc_dist (Point P1, Point P2)
 { return sqrt (((P2.x-P1.x)*(P2.x-P1.x))+((P2.y-P1.y)*(P2.y-P1.y))); }
```

1.3 Solución al ejercicio utilizando clases

1.3.1 Versión 1. Declaración de la función dentro de la clase y definición fuera de ella Utilización del operador de acceso ::

```
// ficha 9c
/* con clases. 1ª forma.*/
#include <iostream.h>

//clase para los objetos tipo punto
class Point
{
public:
int  x, y; //datos tipo entero

Point () {cin>>x; cin >>y;}//constructor sin argumento,para inicialización
Point (int a, int b) {x=a; y=b;} //constructor con 2 argumentos

~Point() {} /destructor

/*declaración dentro de la clase de la función cálculo distancia entre dos puntos.
Distancia entre un punto(argumento tipo punto)en referencia al punto A */
```

```
double calc_dist (Point A);
};

//programa  principal
void main (void)
{
//inicializamos tres puntos
Point P1 (1,1);
Point P2 (4,2);
Point P3 ();

//cálculo de distancia del punto P2 al P1 y escritura
double dist =  P1.calc_dist (P2);
cout << "La distancia es:"<<dist;

}

//definición de la función calculo de distancia fuera de la clase
//utilización del operador de acceso ::

double  Point :: calc_dist (Point A)
 { return sqrt (((A.x-x)*(A.x-x))+((A.y-y)*(A.y-y))); }
```

1.3.2 Versión 2. Declaración y definición de la función fuera de la clase. Datos públicos

```
// ficha 9c
/* con clases. 2ª forma.*/
#include <iostream.h>

//clase para los objetos tipo punto
class Point
{
public:
int  x, y; //datos tipo entero

Point () {cin>>x; cin >>y;}//constructor sin argumento. Inicialización
Point (int a, int b) {x=a; y=b;} //constructor con 2 argumentos

~Point() {} /destructor
};

/*declaración función cálculo distancia entre dos puntos fuera de la
clase y antes del programa principal. Función con dos argumento tipo punto*/
double calc_dist (Point P1, Point P2);

//programa  principal
void main (void)
{
//inicializamos tres puntos
Point P1 (1,1);
Point P2 (4,2);
Point P3 ();

// cálculo de distancia del punto P2 al P1 y escritura
double dist = calc_dist ( P1, P2);
cout << "La distancia es:"<<dist;

}

/*definición de la función calculo de distancia fuera de la clase. Como ha sido
declarada también fuera de la clase no necesita el operador de acceso ::*/
```

```
/*Si los datos de la clase fuesen privados no se puede acceder a ellos desde una
función externa a la clase. hay que acceder a través de una función que previamente
hallamos definido en la clase. */
double calc_dist (Point P1, Point P2)
 {
 return sqrt (((P2.x-P1.x)*(P2.x-P1.x))+((P2.y-P1.y)*(P2.y-P1.y)));
 }
```

1.3.3 Versión 3. Declaración y definición de la función fuera de la clase. Datos privados

```
// ficha 9c
/* con clases. 3ª forma.*/
#include <iostream.h>

//clase para los objetos tipo punto
class Point
{
private: //acceso privado
int  x, y; //datos tipo entero

public: //acceso público

Point (int a, int b) {x=a; y=b;} //constructor con 2 argumentos

~Point() {} /destructor

Point(Point& a) //constructor copia
{x=a.get_x(); y =a.get_y();}

//funciones para acceder a los datos privados
int get_x() {return x};
int get_y() {return y};
};

/*declaración función cálculo distancia entre dos puntos
fuera de la clase y antes del programa principal. Función con dos argumentos tipo
punto*/
double calc_dist (Point P1, Point P2);

//programa  principal
void main (void)
{
//inicializamos tres puntos
Point P1 (1,1);
Point P2 (4,2);

//cálculo de distancia del punto P2 al P1 y escritura
double dist = calc_dist ( P1, P2);
cout << "La distancia es:"<<dist;

}

//definición de la función calculo de distancia
/*Para poder acceder a los datos privados de la clase desde una función externa a
la misma, hay que accederlo a través de las funciones previamente definidas en la
clase para ello. */

double  calc_dist (Point P1, Point P2)
 {
 return sqrt (((P2.get_x())-(P1.get_x()))*((P2.get_x())-(P1.get_x()))+
((P2.get_y())-(P1.get_y()))*((P2.get_y())-(P1.get_y()))); }
```

Ficha 10: Herencia

1 Objetivos generales

- Conocer la potencia de la herencia.

2 Código de trabajo

```
Ficha 10
/* Herencia */

#include <iostream.h>

class Complex                       //clase de números complejos
{
private :
double real,imag ;                  // datos
public:
Complex (double a, double b)        //constructor
{ real=a ; imag =b ; }
~Complex ()                         //destructor
{}
Complex (Complex& a)                //constructor copia
{ real = a.get_real() ; imag=a.get_imag();}

Complex& operator = (Complex& m) //operador asignación
{real = m.get_real(); imag = m.get_imag() ; return *this;}

double get_real(void) {return real;}//acceso datos private
double get_imag(void) {return imag;}

friend ostream& operator << (ostream& os, Complex& a)//operador<<
{
os << "COMPLEX=";
os << a.get_real()<<"+"<<a.get_imag()<<"i"<<endl;
return os;
}
};

class MyReal  : public Complex   // clase números reales
                                 // derivada public de Complex
{
public:
MyReal (double a) : Complex (a, 0.) {} //constructor
};
```

```
031    int main (void)            //programa principal
032    {
033    Complex a(1.,1.);          // objeto tipo Complex
034    MyReal b(2.);              // objeto tipo MyReal
035    Complex c= b;       //asignación objeto MyReal a objeto Complex
036    MyReal d=a;         //asignación objeto Complex a objeto MyReal. Error
037    cout << a <<b;             // escritura por pantalla
038    return 0;
039    }
```

3 Conceptos

3.1 Qué representa la herencia

Gracias a la herencia se pueden construir nuevas clases a partir de otras, añadiendo nuevas características sin tener que reprogramar toda la clase de nuevo. La clase de la que se hereda se denomina *clase base*, y la clase que hereda se denomina *clase derivada.* Una clase derivada puede ser a su vez derivada de otra clase, convirtiéndose así en una clase base. En el código de trabajo se ha definido una *clase base* (clase *Complex*) (líneas **004**-**025**) y una *clase derivada* de ésta (clase *MyReal*) (líneas **026**-**030**).

Para derivar una clase de otra se escribe dos puntos seguidos por el nombre de la clase base (línea **026**) con un identificador del tipo de acceso (*public, protected* o *private*):

class Derivada : tipo acceso   Base

Se pueden definir objetos de las dos clases tal como se había hecho hasta ahora. En la línea de código **033** se define un objeto de nombre *a* de la clase base tipo *Complex* y se inicializa al valor (1.,1.). En la línea **034** se define el objeto *b* de la clase derivada tipo *MyReal* y se inicializa con el valor (2.).

Una clase derivada no tiene por qué heredar todos los miembros de su clase base, esto será posible en función del acceso y siempre habrá tres excepciones: los constructores, el destructor y los operadores de asignación, ya que estos miembros son los que definen los fundamentos de una clase.

Así, en el código de trabajo anterior se tiene:

Línea de código	*Miembro*
009	constructor clase Complex (base)
011	destructor clase Complex (base)
015	operador de asignación clase Complex (base)
029	constructor clase MyReal (derivada)
sin definir (por defecto)	destructor clase MyReal (derivada)

Cuando se construye un objeto de una clase derivada, se llama en primer lugar al constructor de la base, después se construyen todos los miembros de esa clase derivada y luego se llama al constructor de la clase derivada. Con los destructores el proceso se invierte; para destruir un objeto de la clase derivada, primero se llama al destructor de la clase derivada, luego se destruyen los miembros de la clase derivada y por último se llama al destructor de la clase base.

El hecho de que la clase derivada herede de la clase base quiere decir que hace una copia de todos sus miembros públicos o privados (excepto los ya indicados). Podemos por tanto, llamar desde la clase derivada a los métodos y atributos de la clase base, cuando los atributos heredados de la clase base sean privados se llamará a los métodos de la clase base que trabajen sobre ellos.

Podría decirse, con ciertas matizaciones, que dentro de la clase derivada se ha incluido, como un atributo más, un objeto de la clase base. Esto permite hacer asignaciones entre objetos de una clase derivada a una clase base (línea **035**), pero no el proceso inverso (línea **036**), ya que un objeto de la clase derivada siempre será un objeto de la clase base, pero no al revés. Sobre este tema se insiste en la próxima ficha 13.

3.2 Herencia public, protected y private

Como ya se vio en la ficha 9, el acceso a los miembros de una clase puede ser:

- *private* (privado); sólo se puede acceder a ellos desde la clase base.
- *protected* (protegidos); sólo se puede acceder a ellos desde la clase base donde están definidos y desde las clases derivadas.
- *public* (públicos); se puede acceder a ellos desde cualquier campo.

Al derivar una clase también se puede definir el *tipo de herencia* de la clase base, tal y como se muestra en la línea **026**, las posibildades son:

- herencia *public*; todos los miembros de la clase base que se heredan conservan su nivel de acceso para la clase derivada.
- herencia *protected*; los miembros públicos y protegidos de la clase base pasan a ser miembros *protected* para la clase derivada, y los miembros privados de la clase base pasan a ser inaccesibles para la clase derivada y todas las clases descendientes.
- herencia *private*; los miembros públicos y protegidos de la clase base pasan a ser miembros privados para la clase derivada y los miembros privados de la clase base pasan a ser inaccesibles para la clase derivada y todas sus descendientes.

Nótese que, aunque los miembros privados de la clase base no sean accesibles por la clase derivada, se heredan para que los métodos heredados puedan trabajar sobre ellos.

Resumiendo, las especificaciones de acceso a clase base y sus conversiones son:

	clase BASE	***clase DERIVADA***
public	públicos	públicos
	protegidos	protegidos
	privados	INACCESIBLES
protected	públicos	protegidos
	protegidos	protegidos
	privados	INACCESIBLES
private	públicos	privados
	protegidos	privados
	privados	INACCESIBLES

4 Ejercicios

4.1 Definir una clase para nudos en línea, plano y espacio.

4.1.1 Versión primera

```
// Ficha 10
/* Definir una clase para nudos en linea, plano y espacio. Opción 1. */

# include <iostream.h>
# include <math.h>

Class Espacio             // clase para puntos en el espacio
 {
  public:double x,y,z;           // datos
  Espacio (double a, double b, double c) { x=a, y=b; z=c;}// constructor
  ~ Espacio (){}                                          // destructor

  Espacio (Espacio& a)                              //constructor copia
  { x = a.x; y= a.y; z= a.z;}

  Espacio& operator = (Espacio& m)                  //operador asignación
  { x = m.x; y = m.y; z = m.z;
  return *this;}

  friend Espacio operator + (Espacio& a,Espacio& b)  //operador suma
  {
   Espacio S(0.,0.,0.);
   S.x=a.x+b.x; S.y=a.y+b.y; S.z=a.z+b.z;
   return S;
  }

  friend ostream& operator << (ostream& os, Espacio& a)//operador <<
  {
  os <<" Espacio=";
  os << a.x <<","<< a.y<<","<<a.z<<endl;
  return os;
  }
  };

 class Plano : public Espacio    //clase para puntos en el plano
                                //derivada public de Espacio
 {
  public:
  Plano (double x,double y): Espacio (x,y,0.) {}           //constructor

   friend ostream& operator << (ostream& os, Plano& a)     //operador <<
  {
  os <<" Plano =";
  os << a.x <<","<< a.y<<endl;
  return os;
  }
 };

 class Linea : public Plano      //clase para puntos linea
                                 //derivada public de Plano
 {
  public:
  Linea (double x) : Plano (x,0.){}                        //constructor

  friend ostream& operator << (ostream& os, Linea& a)      //operador <<
  {
```

```
  os <<" Linea =";
  os << a.x <<endl;
  return os;
  }
 };

double Distancia (Espacio A, Espacio B);
 /*declaración función distancia fuera de la clase y antes del programa
principal.Dos argumentos Espacio.Distancia entre dos puntos Espacio */.

int main (void)           //programa principal
{
Espacio a (1.,1.,1.);     //objeto tipo Espacio
cout <<"a="<<a<<endl;     //escritura por pantalla

Espacio d=a;              //asignación de un objeto espacio a otro
cout <<"d="<<d<<endl;     //escritura por pantalla

Plano b(2.,1.);           //objeto tipo Plano
cout << "b="<< b<<endl;   //escritura por pantalla

Espacio l= a+b;           //definición objeto Espacio como suma de  objeto Espacio y
Plano
cout << "l="<<l<<endl;    //escritura por pantalla

Espacio k=a+a;//definición objeto Espacio como suma de dos objetos Espacio
cout<<"k="<< k<< endl;    //escritura por pantalla

Linea m(8.);              //objeto Linea
cout << "m="<< m<<endl;   //escritura por pantalla

Espacio n= b+m;//definición objeto Espacio como suma de objeto Plano Linea
cout << "n="<<n<<endl;    //escritura por pantalla

double p;                 //objeto double
p = Distancia (a,b);      //distancia entre un objeto Espacio y uno Plano
cout << "p="<<p<<endl;    //escritura por pantalla

return 0;
}

double Distancia  (Espacio A, Espacio B)//definición función distancia

 {return sqrt ((B.x-A.x)*(B.x-A.x)) + ((B.y-A.y)*(B.y-A.y)) +
 ((B.z-A.z)*(B.z-A.z)));}
```

4.1.2 Versión segunda

```
// Ficha 10
/* Definir una clase para nudos en línea, plano y espacio.Opción 2. */

# include <iostream.h>
# include <math.h>
 class Espacio      //clase para puntos en el espacio
 {
  public:double x,y,z;
  Espacio (double a, double b, double c) { x=a, y=b; z=c;}
  ~ Espacio (){}

  Espacio (Espacio& a)
  { x = a.x; y= a.y; z= a.z;}
```

```
  Espacio& operator = (Espacio& m)
  { x = m.x; y = m.y; z = m.z;
  return *this;}

  friend Espacio operator + (Espacio& a,Espacio& b)
  {
   Espacio S(0.,0.,0.);
   S.x=a.x+b.x; S.y=a.y+b.y; S.z=a.z+b.z;
   return S;
  }

  friend ostream& operator << (ostream& os, Espacio &a)
  {
  os <<" Espacio=";
  os << a.x <<","<< a.y<<","<<a.z<<endl;
  return os;
  }

  double Distancia (Espacio A);
 /*declaración función distancia dentro de la clase.Un argumento Espacio,
  (distancia relativa a una referencia tipo Espacio)*/
  };

 class Plano : public Espacio    //clase puntos en el plano
 {
  public:
  Plano (double x,double y): Espacio (x,y,0.) {}

  friend ostream& operator << (ostream& os, Plano &a)
  {
  os <<" Plano=";
  os << a.x <<","<< a.y<<endl;
  return os;
  }
  };

 class Linea : public Plano      //clase puntos linea
 {
  public:
  Linea (double x) : Plano (x,0.){}

  friend ostream& operator << (ostream& os, Linea &a)
  {
  os <<" Linea=";
  os << a.x <<","<< a.y<<endl;
  return os;
  }
 };

int main (void)            //programa principal
{
Espacio a (1.,1.,1.);
cout <<"a="<<a<<endl;

Espacio d=a;
cout <<"d="<<d<<endl;

Plano b(2.,1.);
cout << "b="<< b<<endl;

Espacio l= a+b;
cout << "l="<<l<<endl;

Espacio k=a+a ;
cout<<"k="<< k<< endl;
```

```
Linea m(8.);
cout << "m="<< m<<endl;

Espacio n= b+m;
cout << "n="<<n<<endl;

double p;
p = a. Distancia (b);
cout << "p="<<p<<endl;

return 0;
}
//Definición función distancia. Operador de campo ::
double Espacio :: Distancia  (Espacio A)
 {return sqrt (((A.x-x)*(A.x-x))+((A.y-y)*(A.y-y))+((A.z-z)*(A.z-z)));}
```

4.2 Definir cuadrilátero y cuadrado. Constructores.

Solución al ejercicio.

```
// Ficha 10
/* Definir cuadrilàtero y cuadrado. Constructores */

# include <iostream.h>
# include <math.h>

class Point  //clase punto
 {
  public:
  double x,y;
  public:
  Point(){}
  Point (double a, double b) { x=a; y=b;}
  ~ Point () {};

  Point (Point& P)
  {x=P.x; y= P.y;}

  Point& operator = (Point& P)
  { x=P.x; y=P.y;
   return * this; }

  friend Point operator + (Point& a, Point& b)
  {
  Point S(0.,0.);
  S.x=a.x+b.x; S.y=a.y+b.y;
  return S;
  }
  friend ostream& operator << (ostream& os , Point& P)
  {
  os << "("<<P.x<<","<<P.y<<")";
  return os;
  }
  };

  class Cuadrilatero      //clase cuadrilátero con datos tipo Point
 {
  public:

  Point P1,P2,P3,P4;
  public:
```

```
  Cuadrilatero (){}
  Cuadrilatero (Point A1, Point A2, Point A3, Point A4)
  {P1=A1 ; P2=A2; P3=A3; P4=A4 ;}

  ~ Cuadrilatero () {}

  Cuadrilatero (Cuadrilatero& C)
  {P1 = C.P1; P2= C.P2; P3= C.P3; P4=C.P4;}

  Cuadrilatero& operator = (Cuadrilatero& D)
  { P1 = D.P1; P2= D.P2; P3= D.P3; P4=D.P4;
  return *this;}
 friend ostream& operator << (ostream& os, Cuadrilatero& C)
  {
  os<<"Cuadrilatero=";
  os <<C.P1<<"-"<< C.P2<<"-"<<C.P3<<"-"<< C.P4;
  return os;
  }
  };

 class Cuadrado : public Cuadrilatero
       /* clase cuadrado derivada public de cuadrilatero */
 {
  public:
  Cuadrado (Point P1, Point P3): Cuadrilatero (P1,P2,P3,P4)
   {
   P2.x=P1.x+P3.x; P2.y=P1.y; P4.x=P1.x; P4.y=P1.y +P3.y;
   }

   friend ostream& operator << (ostream& os, Cuadrado& C)
  {
  os <<"Cuadrado=";
  os <<C.P1<<"-"<<C.P2<<"-"<<C.P3<<"-"<<C.P4;
  return os;
 }
  };

double Longitud (Point P1, Point P2);

/*declaración función longitud fuera de la clase Point.
Dos argumentos tipo Point,distancia entre dos puntos*/

double Perimetro (Cuadrilatero C);

/* declaración función  perímetro,un argumento tipo Cuadrilatero */

int main (void)     //programa principal
{
Point P1(1.,1.);
Point P2(2.,1.);
Point P3(2.,2.);
Point P4(1.,2.);

cout<<"Primer punto  ="<<P1<<endl;
cout<<"Segundo punto ="<<P2<<endl;
cout<<"Tercer punto  ="<<P3<<endl;
cout<<"Cuarto punto  ="<<P4<<endl;

Cuadrilatero C (P1, P2, P3,P4);

cout <<C<<endl;

Cuadrado K (P1,P3);

cout <<K<<endl;
```

```
double T1;
T1= Perimetro (C);
cout <<"Perimetro Cuarilatero="<<T1<<endl;

double T2;
T2 = Perimetro (K);
cout <<"Perimetro Cuadrado="<<T2<<endl;
return 0;
}

double Longitud  (Point A , Point B)    //definición función longitud

 { return  sqrt (((B.x-A.x)*(B.x-A.x)) + ((B.y-A.y)*(B.y-A.y))); }

double Perimetro (Cuadrilatero C)       //definición función perímetro

  {return ((Longitud (C.P1,C.P2)) + (Longitud (C.P2,C.P3)) +
  (Longitud (C.P3, C.P4)) + (Longitud (C.P4,C.P1))); }
```

4.3 Definir clases para los distintos triángulos tipo que existen. Establecer una jerarquía y constructores.

Solución al ejercicio.

```
// Ficha 10
/* Definir clases para los distintos triángulos que existen.
Establecer una jerarquia y constructores. */

# include <iostream.h>
# include <math.h>

class Point  //clase punto
 {
  public:
  double x,y;

 public:
  Point (){}
  Point (double a, double b) { x=a; y=b;}
  ~ Point () {};

  Point (Point& P)
  {x=P.x; y= P.y;}

  Point& operator = (Point& P)
  { x=P.x; y=P.y;
   return * this; }

  friend Point operator + (Point& a, Point& b)
  {
  Point S(0.,0.);
  S.x=a.x+b.x; S.y=a.y+b.y;
  return S;
  }
  friend ostream& operator << (ostream& os , Point& P)
  {
  os << "("<<P.x<<","<<P.y<<")";
  return os;
  }
   };
```

```
 class Segment            //clase segmento con datos tipo Point
 {
 public:
 Point P1,P2;

 public:
   Segment () {}
   Segment (Point A1, Point A2)
 { P1=A1; P2= A2;}

 ~ Segment () {}

 Segment& operator = (Segment& S)
 {P1= S.P1, P2= S.P2;
 return *this;}
   friend ostream& operator << (ostream& os, Segment& S)
 {
 os<< "("<< S.P1<<")"<<""-""<<"("<<S.P2<<")";
 return os;
 }
  double Longitud ()  /*declaración y definición función longitud
                        dentro de la clase. Sin argumento*/
{ return sqrt(((P2.x-P1.x)*(P2.x-P1.x))+((P2.y-P1.y)*(P2.y-P1.y)));}
 };

 class Trectangulo   //clase triangulo rectangulo con datos tipo Segment
{
 public:
 Segment S1,S2,S3;

 public:
 Trectangulo (Segment N1, Segment N2, Segment N3)
 { S1=N1 ; S2=N2; S3=N3;}

 ~ Trectangulo () {}

 Trectangulo (Trectangulo& T)
 { S1 = T.S1; S2= T.S2; S3= T.S3;}

 Trectangulo& operator = (Trectangulo& T)
 { S1 = T.S1; S2= T.S2; S3= T.S3;
 return *this;}

 friend ostream& operator << (ostream& os, Trectangulo& T)
 {
 os <<" Triangulo=";
 os << T.S1<<"/"<< T.S2<<"/"<<T.S3<<endl;
 return os;
 }

 double Perimetro () /*declaración y definición función perímetro dentro
                      de la clase. Sin argumento */
{ return ((S1.Longitud())+(S2.Longitud())+(S3.Longitud()));}
 };

class Tisosceles:  public Trectangulo  // clase triangulo isosceles
                        //derivado público del triangulo rectangulo
{
 public:
 Tisosceles (Segment S1, Segment S2): Trectangulo (S1,S2,S3)
  {
  (S3.Longitud())== (S2.Longitud());
  }
```

```
  friend ostream& operator << (ostream& os, Tisosceles& T)
  {
  os <<" Triangulo =";
  os << T.S1<<"/"<< T.S2<<endl;
  return os;
  }
   };


 class Tequilatero:  public Tisosceles //clase triangulo equilátero
                            //derivado público de triangulo isósceles
 {
  public:
  Tequilatero (Segment S1): Tisosceles (S1,S2)
     { (S2.Longitud())== (S1.Longitud());
        (S3.Longitud())== (S1.Longitud());}
 };

 int main (void)          //programa principal
{

double P1x, P1y, P2x, P2y, P3x, P3y;

cout <<"introduzca coordenadas del punto 1="<<endl;
cin >>P1x, P1y;
cout <<"introduzca coordenadas del punto 2="<<endl;
cin >>P2x, P2y;
cout <<"introduzca coordenadas del punto 3="<<endl;
cin >>P3x, P3y;

Point P1(P1x, P1y);
Point P2(P2x, P2y);
Point P3(P3x, P3y);

Segment S1 (P1,P2);
Segment S2 (P2,P3);
Segment S3 (P3,P1);
Trectangulo T1 (S1,S2,S3);
Tisosceles T2 (S1,S2);
Tequilatero T3 (S1);

double p1, p2, p3;

p1 = T1. Perimetro ();    // calculo  perímetro de objeto tipo Trectangulo
p2 = T2. Perimetro ();    // cálculo perímetro de objeto tipo Tisosceles
p3 = T3. Perimetro ();    // cálculo perímetro de objeto tipo Tequilatero

cout <<T1;                // escritura por pantalla de un Trectangulo
cout << "Perimetro ="<< p1<<endl; //escritura por pantalla de un double

cout <<T2;
cout << "Perimetro ="<< p2<<endl;

cout <<T3;
cout << "Perimetro ="<< p3<<endl;

return 0;
}
```

5 Ejemplo

Se define la clase para matrices y de sus propiedades se hereda una clase para vectores. La jerarquía establece, pues, que los vectores sean derivados de la clase matriz.

```
#include <iostream.h>
#include <stdlib.h>
#define MAX_SIZE 25

class Matrix
{
public:
  int mfil;  int ncol;
  double s[MAX_SIZE][MAX_SIZE];

public:
  Matrix (void) {ncol=0; mfil=0;}
  Matrix (int m, int n)
  {
        if (m<MAX_SIZE && n<MAX_SIZE)
        { mfil=m; ncol=n; for (int i=0; i<mfil; i++)
             for (int j=0; j<ncol; j++) s[i][j]=0.;}
        else
   {cerr << "la dimension maxima es " << MAX_SIZE
         <<" x " << MAX_SIZE << endl; exit(1);}
  }

 void InicializarMatrix (void);

 Matrix (Matrix& m);
 Matrix operator = (Matrix m);


 ~Matrix() { mfil=0; ncol=0; }

  friend ostream& operator << (ostream& os, Matrix& m);
};

class MatrixSimetrica : public Matrix
{
public:

      MatrixSimetrica (int m) : Matrix (m,m) {}
      ~MatrixSimetrica  () {}
};

class Vector : public Matrix
{
public:
      Vector (int m) : Matrix (1,m) {}
      ~Vector() {}
};

int main()
{
 Matrix A(2,2); A.InicializarMatrix(); cout << "matriz A = " << A << endl;
 MatrixSimetrica B(2); B.InicializarMatrix();
 cout << "matriz B = " << B << endl;
 Vector C(3); C.InicializarMatrix(); cout << "vector C= " << C << endl;
 return 0;
}


Matrix::Matrix(Matrix& m)
```

```
{
 mfil=m.mfil; ncol=m.ncol;
 for (int i=0; i<mfil; i++) for (int j=0; j<ncol; j++) s[i][j]=m.s[i][j];
}



Matrix Matrix::operator = (Matrix m)
{
  Matrix Temp(m.mfil,m.ncol);
  for (int i=0; i<mfil; i++) for (int j=0; j<ncol; j++) Temp.s[i][j]=m.s[i][j];
  return Temp;
}

void Matrix:: InicializarMatrix (void)
{
  for (int i=0; i<mfil; i++) for (int j=0; j<ncol; j++)
       { cout << "coeficiente (" << i << "," << j << ")="; cin >> s[i][j]; }
 return;
}


ostream& operator << (ostream& os, Matrix& m)
{
 os << "Matrix ++++ "; if (!m.mfil && !m.ncol) os<<"void";
 for (int i=0; i<m.mfil; i++)
 {
  os <<endl;
  for (int j=0; j<m.ncol; j++)
        os << m.s[i][j] <<" " ;
  }
 os << "\n";
 return os;
}
```

Ficha 11: Polimorfismo

1 Objetivos generales

- Para ciertos autores la programación de objetos se fundamenta en tres elementos: las clases, la herencia y el polimorfismo. Es el turno del último pilar, el polimorfismo. Parafraseando a los clásicos, el nuevo vino de viejas viñas.

2 Código de trabajo

```
// Ficha 11
/* Hacer cosas nuevas con el mismo nombre, mantener la coherencia abstracta
*/
#include <iostream.h>

class complex//clase de numeros complejos
{
public :
double real , imag ;

 public:
complex (double a, double b)
{ real=a ; imag =b ; }

~complex ()
{}

complex (complex& a)
{ real = a.get_real() ; imag=a.get_imag();}

complex& operator = (complex& m)
{real = m.get_real(); imag = m.get_imag() ; return *this;}

double get_real(void) {return real;}
double get_imag(void) {return imag;}

friend ostream& operator << (ostream& os, complex& a)      //definicion
operador <<
 {
  os << "Complex=";
  os << a.get_real()<<"+"<<a.get_imag()<<"i"<<endl;
  return os;
  }
};

class Real : public complex      /*clase     numeros     reales     derivada
publicamente de la clase de numeros complejos*/
```

```
027    {
028    public:
029    Real (double a) : complex (a, 0.){}

030    friend ostream& operator << (ostream& os, Real& a)
      /*redefinicion  operador <<*/
031     {
032      os << "Real=";
033      os << a.get_real()<<endl;
034      return os;
035     }
036    };
037    int main (void)           //programa principal
038    {
039    Real a(2.);  //objeto clase Real
040    cout<<a<<endl;      //escritura por pantalla de un numero real
041    Complex b=a;
042     cout<<b<<endl;
043     return 0;
044    }
```

3 Conceptos

3.1 Polimorfismo; hacer cosas nuevas con el mismo nombre. Mantener la coherencia abstracta

El *polimorfismo* se define como la posibilidad de utilizar una misma llamada a un método. Es decir, ser capaces de utilizar el mismo nombre para una función. El ámbito de actuación es doble; por un lado usar el mismo nombre en funciones de la propia clase, y además utilizar la misma llamada en clases derivadas, pero actuando de distinta forma en cada una de ellas.

El primer concepto se asume de forma sencilla; en la ficha 9 se hacía mención a la posibilidad de definir diferentes constructores. Pues bien, todos ellos tienen el mismo nombre, distintos argumentos y, lógicamente, diferentes inicializaciones de las variables.

El segundo concepto es más profundo, permite mantener coherencia en el código. En una clase derivada de otra clase base, se pueden clasificar los métodos según:

1. *Métodos heredados*: funciones definidas en la clase base y heredadas por la clase derivada, sin necesidad de definirlas en esta última.

2. *Métodos añadidos*: funciones añadidas en la clase derivada que no fueron definidas en la clase base.

3. *Métodos redefinidos*: funciones definidas en la clase base y redefinidas con el mismo nombre en la clase derivada. Este último caso produce el polimorfismo y se utilizará cuando una función heredada no funcione correctamente o sea necesario añadirle algo.

En el código de trabajo se han definido dos clases; la clase *complex* (clase base) (líneas **004-025**) y la clase *Real* (clase derivada publicamente de la clase complex) (líneas **026-036**). En la clase base se ha definido el operador de escritura por pantalla << para números complejos (líneas **019-024**), y en la clase derivada se ha redefinido para números reales (líneas **030-035**) (polimorfismo del operador <<). Esto permite acceder a este operador mediante un objeto de la clase derivada (línea **040**); la escritura por pantalla será la indicada en la redefinición del operador de escritura. Sin embargo, si se accediese a este operador mediante un objeto de la clase base (línea **042**), por pantalla se obtendría la escritura de un número complejo.

En el código de trabajo se ha utilizado el polimorfismo para el operador de asignación <<; evidentemente se podría haber extendido la idea a algún método compartido. El ejemplo no hace perder generalidad y permite comprender las ideas básicas.

4 Ejercicios

4.1 ¿Qué sucede con la suma de los reales si se hereda de los complejos?

Solución al ejercicio.

```
// Ficha 11
/* Suma de los reales si se heredan de los complejos */

#include <iostream.h>

class complex       //clase de números complejos
{
 public :
 double real,imag ;

 public:
 complex (){}
 complex (double a, double b)
  { real=a ; imag =b ; }
 ~complex ()
  {}

complex (complex& a)
{ real = a.real ; imag=a.imag;}

complex& operator = (complex& m)
{real = m.real; imag = m.imag ; return *this;}

friend ostream& operator << (ostream& os, complex& a)
{
os << "Complex=";
os << a.real<<"+"<<a.imag<<"i"<<endl;
return os;
}

friend complex operator + (complex& a, complex& b)
{
complex c(0,0);
c.real = a.real + b.real;
c.imag = a.imag + b.imag;
return c;
}
};

class Real : public complex            //clase de números reales
{
public:
Real (double a) : complex (a,0.) {}
};

int main (void)           // programa principal
{
 complex a (1.,1.);
 Real b (2.);
 Real c (8.);
```

```
 complex s1;
 s1 = a+b;   // suma de un número complejo y uno real
 complex s2;
 s2 = b+c;   //suma de dos números reales utilizando el operador +    //definido en
             la clase base, ya que la clase derivada lo ha  //heredado.
 cout << s1<<endl;
 cout << s2<<endl;

 return 0;
}
```

4.2 Funciones de cálculo de áreas en las clase de polígonos anteriores.

Solución al ejercicio.

```
// Ficha 11
/* Funciones de cálculo de áreas en las clases de polígonos */

#include <iostream.h>

class Rectang       //clase rectangulo
{
 public :
 double b,h ;

 Rectang (){}
 Rectang (double x, double y)
  { b=x ; h=y; }
 ~Rectang() {}

 Rectang (Rectang& R)
 { b=R.b ; h=R.h;}

 Rectang& operator = (Rectang& T)
 {b=T.b; h=T.h ; return *this;}

 friend double Area (Rectang& a)//declaración y definición de
                           //función area dentro de clase base
 {return ((a.b)*(a.h));}
};

class Cuadrad : public Rectang   //clase cuadrado derivada publicamente
                           //de clase rectangulo
{
public:
 Cuadrad (double l): Rectang (l,l) {}
};

int main (void)            // programa principal
{
Rectang A (1.,2.);
Cuadrad B (3.);
cout<< "El área del Rectángulo es: "<< Area (A)<<endl;
//area de un rectángulo
cout<< "El área del Cuadrado es: "<< Area (B)<<endl;
//area de un cuadrado, utilizando función heredada
return 0;
}
```

5 Ejemplo

En este caso se redefinen operaciones y funciones para las clases derivadas; obsérvese que en tiempo de ejecución el programa sabe cuál es la función que debe tomar y reconoce si la clase es derivada o base.

```
#include <iostream.h>
#include <stdlib.h>
#define MAX_SIZE 25

class Matrix
{
public:
  int mfil;  int ncol;
  double s[MAX_SIZE][MAX_SIZE];

public:
  Matrix (void) {ncol=0; mfil=0;}
  Matrix (int m, int n)
  {
        if (m<MAX_SIZE && n<MAX_SIZE)
             { mfil=m; ncol=n; for (int i=0; i<mfil; i++)
                   for (int j=0; j<ncol; j++) s[i][j]=0.;}
        else
             {cerr << "la dimension maxima es " << MAX_SIZE
             <<" x " << MAX_SIZE << endl; exit(1);}
  }

 void InicializarMatrix (void);

 Matrix (Matrix& m);
 Matrix operator = (Matrix m);


 ~Matrix() { mfil=0; ncol=0; }

  friend ostream& operator << (ostream& os, Matrix& m);
};

class MatrixSimetrica : public Matrix
{
public:

      MatrixSimetrica (int m) : Matrix (m,m) {}
      ~MatrixSimetrica  () {}

   void InicializarMatrix (void);

};

class Vector : public Matrix
{
public:
      Vector (int m) : Matrix (1,m) {}
      ~Vector() {}
      void InicializarVector (void) { Matrix::InicializarMatrix(); }

  friend ostream& operator << (ostream& os, Vector& m);

};

int main()
{
```

```
 Matrix A(2,2); A.InicializarMatrix();
 cout << "matriz A = " << A << endl;
 MatrixSimetrica B(2); B.InicializarMatrix();
 cout << "matriz B = " << B << endl;
 Vector C(3); C.InicializarVector();
 cout << "vector C= " << C << endl;
 return 0;
}

Matrix::Matrix(Matrix& m)
{
 mfil=m.mfil; ncol=m.ncol;
 for (int i=0; i<mfil; i++) for (int j=0; j<ncol; j++) s[i][j]=m.s[i][j];
}

Matrix Matrix::operator = (Matrix m)
{
  Matrix Temp(m.mfil,m.ncol);
  for (int i=0; i<mfil; i++) for (int j=0; j<ncol; j++) Temp.s[i][j]=m.s[i][j];
  return Temp;
}

void Matrix:: InicializarMatrix (void)
{
  for (int i=0; i<mfil; i++) for (int j=0; j<ncol; j++)
       { cout << "coeficiente (" << i << "," << j << ")="; cin >> s[i][j]; }
 return;
}

ostream& operator << (ostream& os, Matrix& m)
{
 os << "Matrix ++++ "; if (!m.mfil && !m.ncol) os<<"void";
 for (int i=0; i<m.mfil; i++)
 {
  os <<endl;
  for (int j=0; j<m.ncol; j++)
        os << m.s[i][j] <<" " ;
  }
 os << "\n";
 return os;
}


void MatrixSimetrica:: InicializarMatrix (void)
{
  for (int i=0; i<mfil; i++)
       for (int j=i; j<ncol; j++)
       { cout << "coeficiente (" << i << "," << j << ")=";
         cin >> s[i][j]; s[j][i]=s[i][j]; }
 return;
}


ostream& operator << (ostream& os, Vector& m)
{
 os << "Vector ++++ "; if (!m.mfil && !m.ncol) os<<"void";
 for (int i=0; i<m.mfil; i++)
 {
  os <<endl;
  for (int j=0; j<m.ncol; j++)
        os << m.s[i][j] <<" " ;
  }
 os << "\n";
 return os;
}
```

Ficha 12a: Punteros. Parte I

1 Objetivos generales

- Conocer qué son los punteros y cómo funcionan. Gestión de la memoria.
- Declaración e inicialización. Operadores de referencia y de indirección.

2 Código de trabajo

```
// Ficha 12a
// Operador &  //

include <iostream.h>

int main (void)
{
int x=1;      //declaración de variable entera , de nombre x ,
       //inicializada al valor 1
int *ip;      //declaración de puntero, de nombre ip, de enteros

ip=&x;        // direccion de ip = dirección de x.
       //(el puntero p apunta a la variable x)

cout <<"la variable x = "<<x<< " y esta en la posición de memoria
"<<ip<<endl;
return 0;
}
```

3 Conceptos

3.1 Punteros. Declaración e inicialización

Los punteros son un maravilloso instrumento para optimizar el código en tiempo de ejecución y en gestión de memoria. A nuestro modesto entender, los punteros no son un tipo de dato, sino una técnica de programación. Por ello se han dejado para más adelante, porque dentro del lenguaje juegan el papel de sintaxis avanzada y no de palabras o de estructuras básicas. Cuando uno aprende un idioma empieza con estructuras simples y con palabras sencillas, sólo en los cursos avanzados se perfecciona el lenguaje. Ésta es la motivación de haberlos introducido tan tarde. Por consiguiente, en las próximas líneas realizaremos un pequeño salto atrás para luego volver hacia delante con mayor impulso.

Un puntero es una variable que contiene la dirección de otra variable. ¿Y esto qué sentido tiene? A veces, es conveniente imaginar la memoria del ordenador como un mueble con numerosos cajones etiquetados con números: 1, 2, 3, etc., cada uno de ellos representa una variable y dentro del cajón se halla su valor. El mueble está guardado por un celador que tienen una pequeña ventanilla de atención al público. Supongamos que se desea conocer el valor de una determinada variable, para ello sería necesario ir al celador y decirle que estamos interesados en la variable *x*. El celador conoce cuál es su cajón asociado, por ejemplo el 25, lo abre y extrae el número que guarda en su interior. La variable es *x*, el puntero es la etiqueta del cajón, el 25.

La pregunta inmediata es, ¿por qué necesito punteros si ya conozco mis variables por su nombre? Simplemente para programar de forma más eficiente, en esta ficha y las siguientes se observará la ventaja del uso de punteros.

El puntero permite dos operaciones con respecto a una variable:
- obtener directamente el valor de la variable, operador de indirección (*).
- obtener la dirección de la variable, operador de referencia (&).

La sintaxis para la declaración de punteros en C++ es:
Tipo * NombrePuntero;

En la línea de código **007** se ha definido un puntero de enteros de nombre *ip*.

3.2 Operador &

El operador & se denomina *operador de referencia*. Cuando aplicamos este operador a una variable tomamos su dirección; en la línea de código **008** (*ip* = *&x*) hacemos que el puntero *ip* almacene la dirección de *x*, no su valor. Esto se suele expresar diciendo que *ip* apunta a *x*.

Este operador sólo puede ser aplicado a variables y funciones, pero no a expresiones. Por ejemplo, tanto &(x*5) como &(x++) darían un error.

Hay que tener mucho cuidado en no dejar ningún puntero sin inicializar, ya que si no, su dirección quedaría indefinida.

4 Código de trabajo

```
100   // Ficha 12a
101   // operador *  //

102   # include <iostream.h>

103   int main (void)
104   {
105   int x=1; // variable entera , de nombre x, con valor asignado 1
106   int *ip; //puntero, de nombre ip, a enteros

107   ip=&x;   //dirección de ip = dirección de x

108   int y=0 ;// variable entera , de nombre y, con valor asignado 0
109   y=*ip;   //valor de variable y = valor de variable apuntada por ip
               //= valor de variable x
110   y+=1;

111   cout <<"la variable x = "<<x<< " y la variable y = "<<y<<endl;
```

```
112    return 0;
113    }
```

5 Conceptos

5.1 Operador *

El operador * se denomina *operador de indirección*. Cuando se aplica a un puntero nos da el valor de la dirección a la que apuntaba. En la línea de código **109** (*y=*ip*) asignamos a la variable *y* entera el valor de la dirección de *ip*, que es el valor de la variable *x* (línea **105**). El resultado es análogo a una línea de código del tipo *y=x*.

6 Ejercicios

6.1 Leer un par de variables y operar con ellas, a través de la variable y utilizando sus punteros respectivos. Qué significa *ip+=1, ++*ip, (*ip)++., *ip++..etc.

Solución al ejercicio

```
// Ficha 12a
/* Leer un par de variables y operar con ellas,
a traves de la variable y utilizando sus  punteros respectivos */

# include <iostream.h>

int main (void)
{
int x=1;      //variable entera x con valor 1
int *ip;      //puntero ip
int y;        //variable y

ip=&x;        //dirección de ip = dirección de x
*ip+=1;       //aumentamos en uno el valor de la dirección de ip
              //ip = (1+1) = 2
y = *ip;      //valor de y = valor de la dirección de ip = 2

cout <<"el valor de la dirección de ip vale = "<<y<<endl;

return 0;
}
```

7 Ejemplos

Se define un apuntador para poder moverse a lo largo de una matriz. Por lo tanto, se debe poder utilizar tanto la sintaxis con la variable de la matriz como realizar operaciones con un apuntador a ella; el resultado obtenido debe ser el mismo.

```
#include <iostream.h>
#include <stdlib.h>

#define MAX_SIZE 25

int main()
{
 int iMat [MAX_SIZE];
 int size;
```

```
 cout << "escribe las dimensiones de la matriz "; cin >> size;
 if (size > MAX_SIZE)
 { cerr << " tamaño maximo " << MAX_SIZE << " Error" << endl; exit(1); }

 int *MyPointer;
 MyPointer=&iMat[0];

 cout << "entrada de datos " <<endl;
 for (int i=0; i<size; i++)
 {
 cout << " coeficiente (" << i << ")="; cin >> *MyPointer;  MyPointer++;
 }

 MyPointer=&iMat[size-1];
 cout << "matriz ++++++++ " <<endl;
 for (i=0; i<size; i++)
 {
 cout << " " << *MyPointer; MyPointer--;
 }

 return 0;
}
```

Ficha 12b: Punteros. Parte II

1 Objetivos generales

- Punteros, vectores, cadenas y matrices.

2 Código de trabajo

```
200    // Ficha 12b
201    /* Cadenas  (vectores) y punteros */
202    # include <iostream.h>

203    int main (void)
204    {
205    int x[10];    // variable (vector) tipo entero de dimensión 10
206    int *ip;      // puntero de enteros
207    ip=&x [0];    //dirección de ip = dirección de x [0]
// asignamos valores y direcciones de memoria a cada posición del vector
208    for (int i=0 ; i<10 ; i++)
209    {
210    x[i] =i;
211    cout <<"La posicion de memoria"<<ip+i<<endl;
212    cout <<"tiene el valor" << *(ip+i)<<endl;
213    }
214    return 0;
215    }
```

3 Conceptos

3.1 Vectores, punteros y cadenas

En C++ hay una fuerte relación entre los punteros y los vectores. Se puede definir un vector de la forma indicada en la línea de código **205**:

int x[10] ;   donde x es un vector de enteros de dimensión 10

El acceso a las diferentes posiciones del vector se realiza de la forma:

x[0] , x[1] , x[2] , x[3] , x[4] , x[5] , x[6] , x[7] , x[8] , x[9]

tal y como se hace en la línea **210** cuando se inicializan los valores del vector.

También se puede realizar el acceso a través de un apuntador (definido en **206**) e irlo desplazando a lo largo de la memoria. Para ello bastaría con inicializar el puntero en el primer término del vector, tal y como se define en la línea **207**. A continuación moverse a lo largo de las direcciones de memoria incrementando el valor del puntero *ip+i* en la línea **212**. Finalmente, accediendo al valor con el operador de indirección, **(ip+i)* en **212**. Por lo tanto, x[5] (el sexto elemento) sería equivalente a *(x+5) (sacar el valor de la dirección 5 más la inicial).

Con esto queda claro que el operador [ ] se puede expresar según operaciones de punteros aprovechando que en C++ hay una conversión trivial desde T[ ] a T*.

Atención, porque la expresión *ip=& x[0]*, de la línea **207**, puede ser substituida por *ip=x,* ya que x se considera como un puntero a enteros por defecto. En consecuencia, *ip* y *x* son punteros del mismo tipo. No obstante, el vector x es un puntero constante con lo que, recordando que se puede asignar a una variable una constante, pero no al revés, no será válida la expresión *x= ip*.

En las líneas de código **210-212** se asignan valores a las diferentes posiciones de x mediante el operador [] y se accede a ellas mediante un apuntador. Nótese que es posible acceder a los valores de forma consecutiva con el operador **ip++*, pero mucha atención a la precedencia de los operadores, porque el resultado sería distinto si se escribiera *(*ip)++*.

Un caso particular de vector es el vector de *char*, aquello que normalmente recibe el nombre de cadena (*string*). El mecanismo de funcionamiento es el mismo que el señalado anteriormente, pero el tipo de dato es diferente. Se debe destacar que existen en C++ una librería de funciones para trabajar con cadenas, en particular copiarlas *strcp*, compararlas *strcmp*, etc. Todas esas funciones están definidas en el archivo de cabezera *string.h*, pero en esta ficha no se entra en este tipo de detalles que se pueden encontrar en otros libros.

4 Código de trabajo

```
// Ficha 12b
/* Cadenas (matrices) y punteros */

# include <iostream.h>

int main (void)
{
int x[3][4]; // variable (matriz) tipo entero de dimensión 3x3
int *ip;     //puntero de enteros

ip=&x [0][0] ;      // dirección de ip = dirección de x [0][0]

 //asignamos valores y direcciones de memoria a cada posición de la matriz
for (int i=0 ; i<3 ; i++)
    for (int j=0; j<4; j++)
            {
             x[i][j]=i+j;
             cout <<"x ("<<i<<", "<<j<<")=" <<*(ip++)<<"en la
       posicion de memoria"<<ip-1<<endl;
            }

return 0;
}
```

5 Conceptos

5.1 Matrices y punteros

Para definir vectores multidimensionales, en particular de dos dimensiones, se utiliza la sintaxis de la línea **305**. Comentarios paralelos al caso anterior pueden hacerse para el acceso a las posiciones de la matriz mediante el operador [] y la combinación con el operador de indirección y un puntero.
Obsérvense las líneas **307**, **312** y **313**.

Hay que distinguir entre lo que es un puntero a punteros y lo que es un puntero constante (bidimensional). Con ambos se puede conseguir multidimensionalidad, pero el primero permite matrices con filas de diferente longitud y un dimensionamiento dinámico más ágil (ficha 12c); por contra el segundo es muy claro:

```
int  * * ip;      //puntero a puntero de enteros
int   x[ ] [ ]    //puntero constante (bidimensional) a enteros
```

6 Ejercicios

6.1 Operar con matrices y cadenas.

Solución al ejercicio

```
// Ficha 12b
/* Operar con matrices y cadenas */

# include <iostream.h>

int main (void)
{

int x[3];           //vector x de dimensión 3
int y[3][2];        //matriz y de dimensión 3x2
int z[2][3];        //matriz z de dimensión 2x3

int *ipx;           //puntero de enteros ipx
int *ipy;           //puntero de enteros ipy
int *ipz;           //puntero de enteros ipz

ipx=&x [0];         //dirección ipx = dirección x[0]
ipy=&y [0][0];      //dirección ipy = dirección y [0][0]
ipz=&z [0][0];      //dirección ipz = dirección z [0][0]

//Damos valores y posicionamos en memoria el vector x

 for (int i1=0 ; i1<3 ; i1++)
    {
    x[i1]=i1;
    cout <<"x ("<<i1<<")="<<x[i1]<<" en la posicion de     memoria"<<ipx+i1<<endl;
    }
//Damos valores y posicionamos en memoria el vector y

 for (int i2=0 ; i2<3 ; i2++)
       for (int j=0; j<2; j++)
    {
    y[i2][j] =i2+j;
    cout <<"y ("<<i2<<", "<<j<<")="<<y[i2][j]<<" en la posicion de
       memoria"<<ipy++<<endl;
```

```
    }

//Damos valores y posicionamos en memoria el vector z

for (int i3=0 ; i3<2 ; i3++)
      for (int j=0; j<3; j++)
    {
    z[i3][j] =i3+(2*j);
    cout <<"z ("<<i3<<", "<<j<<")="<<z[i3][j]<<" en la posicion de
      memoria"<<ipz++<<endl;
    }

 int M1[1][2];      // declaramos una matriz de dimensiones 1x2
 int M2[2][2];      // declaramos una matriz de dimensiones 2x2

//producto del vector X por la matriz Y

 for (int i4=0 ; i4<1 ; i4++)
      for (int j=0; j<2; j++)
    {
    M1[i4][j] =0;
    }

//producto de la matriz Z por la matriz Y

 for (int i5=0 ; i5<2 ; i5++)
      for (int j=0; j<2; j++)
    {
    M2[i5][j] = 0;
    }

 for (int k1=0; k1<1; k1++)
   for (int i1=0 ; i1<2 ; i1++)
      {
       {
      for (int j1=0; j1<3; j1++)
      M1[k1][i1] += x[j1] * y[j1][i1];
      }
    cout <<"M1 ("<<k1<<", "<<i1<<")="<<M1[k1][i1]<<endl;
    }

 for (int k2=0; k2<2; k2++)
     for (int i2=0 ; i2<2 ; i2++)
      {
       {
      for (int j2=0; j2<3; j2++)
    M2[k2][i2] +=  z[k2][j2] *  y[j2][i2];
          }
    cout <<"M2 ("<<k2<<", "<<i2<<")="<<M2[k2][i2]<<endl;
    }
    return 0;
}
```

7 Ejemplos

En este caso se define una matriz doble y se observa cómo los apuntadores trabajan sobre las direcciones de los diferentes coeficientes.

```
#include <iostream.h>
#include <stdlib.h>

#define MAX_SIZE 25
```

```
int main()
{
 int iMat [MAX_SIZE][MAX_SIZE];
 int mfil,ncol;

 cout << "escribe las dimensiones de la matriz \nfilas=";
 cin >> mfil; cout << "columnas="; cin >> ncol;

 if (mfil > MAX_SIZE || ncol > MAX_SIZE)
 { cerr << " tamaño maximo para filas y columnas"
        << MAX_SIZE << " Error" << endl; exit(1); }

 int *MyPointer;

 cout << "entrada de datos " <<endl;

 int i,j;
 for (i=0; i<mfil; i++)
       {
        MyPointer=&iMat[i][0];
        for (j=0; j<ncol; j++)
        {
        cout << " coeficiente (" << i <<"," << j << ")=";
        cin >> *MyPointer; MyPointer++;
        }
       }

 cout << "matriz ++++++++ " <<endl;
 for (i=0; i<mfil; i++)
       {
        MyPointer=&iMat[i][ncol-1];
        cout << endl;
        for (j=0; j<ncol; j++)
              {cout << " " << *MyPointer; MyPointer--;}
       }

 return 0;
}
```

Ficha 12c: Punteros. Parte III

1 Objetivos generales

- Paso por valor y referencia en funciones.
- Operadores *new* y *delete*.

2 Código de trabajo

```
// Ficha 12c
/* Paso por valor y referencia en funciones */

# include <iostream.h>

/*declaración de funciones para intercambio
 de los valores de las variables */

void swapping1 (int i, int j);          //paso por valor
void swapping2 (int *i, int *j);        //paso por puntero
void swapping3 (int &i, int &j);        //paso por referencia

int main (void)     //programa principal
{
int f;

//valores y direcciones iniciales  de las variables

int x=1, y=2;//definición de variables e inicialización x=1, y=2
int *ix = &x, *iy =&y;//definición de punteros e inicialización
                       // &x = OX352E ,  &y = 0X352A

cout <<"x=" << x <<" direccion x="<< &x << " y="<< y <<
 " direccion y="<< &y <<endl;
cout <<"que función desea aplicar (1,2,3)?"<<endl;
cin >>f;

switch  ( f)
{
case 1:
swapping1(x,y);     //x=1, y=2 , &x = OX352E , &y = 0X352A
cout <<"x=" << x <<" direccion x="<< &x << " y="<< y <<
" direccion y="<< &y <<endl;
break;

case 2:
```

```
420    swapping2(ix , iy); //x=2, y=1 , &x = OX352E , &y = 0X352A
421    cout <<"x=" << x <<" direccion x="<< &x << " y="<< y <<
      "  direccion y="<< &y <<endl;
422    break;

423    default:
424    swapping3(x,y);     //x=2, y=1 , &x = OX352E , &y = 0X352A
425    cout <<"x=" << x <<" direccion x="<< &x << " y="<< y <<
      " direccion y="<< &y <<endl;
426    break;
427    }
428    return 0;
429    }

       /*definición de funciones para intercambio
         de valores de las variables */

430    void swapping1 (int i, int j)    //paso por valor
                                        //(se duplica el valor)
431    {                    //NO cambia el valor original de las variables
432    int temp=i; i=j; j =temp;//NO cambia dirección original variables
433    cout << "i=" << i << " j=" << j <<endl;//lenguaje natural
434    }

435    void swapping2(int *i, int *j)   //paso por puntero
                                        // (solo se duplica la dirección)
436    {                    //SI cambia el valor original de las variables
437    int temp=*i; *i=*j; *j =temp;//NOcambia direcc.original variables
438    cout << "i=" << *i << "j=" << *j <<endl;
439    }
440    void swapping3 (int &i, int &j)  //paso por referencia.
                                        //(Mezcla de 1 y 2)
441    {                                //solo se duplica la direccion.
                                 //Codigo análogo al caso 1
       int temp=i; i=j; j =temp;//SI cambia el valor original variables
443    cout << "i=" << i << "j=" << j <<endl;
444    }             // NO cambia la dirección original de las variables
```

3 Conceptos

3.1 Paso por valor y referencia en funciones

Hasta ahora los argumentos de las funciones eran variables que se utilizaban en el programa. Dicha definición implica que los argumentos de la función se están pasando *por valor*. Por ejemplo, la línea **430** donde se define la función *swapping1* contiene dos argumentos de tipo *int*.

En el *paso por valor* se pasa a la función una copia temporal de la variable, no de su dirección. Esto implica que la función no puede modificar el valor de las variables externas a él, ya que ha sido pasada una copia y no el original. En definitiva, las variables se convierten en locales para el resto del programa. Por lo tanto, se produce un doble fenómeno, se duplica la información y las variables del argumento no se podrán modificar dentro de la función.

Por ejemplo, en el codigo de trabajo se han definido una variables a las que se les ha asignado un valor inicial:

```
408    int x=1, y=2;       //definición de variables e inicialización x=1, y=2
```

Se ha definido también la función *swapping1* a la que se le pasan los argumentos por valor e intercambia sus valores respectivos dentro de la función:

```
void swapping1 (int i, int j)      //paso por valor (se duplica el valor)
{
int temp=i; i=j; j =temp;
cout << "i=" << i << " j=" << j <<endl;
}
```

La llamada a la función será:

```
case 1: //NO cambia el valor original de las variables .Sentencias del lenguaje natural
swapping1(x,y);                    //x=1, y=2
```

Pero si ejecutamos el programa, las variables *x* e *y* no habrán intercambiado sus valores. Esto es así porque, al recibir los argumentos por valor, se crean dos copias temporales que son las que se intercambian dentro de la función, pero al retornar la función se destruyen y por lo tanto no se modifican los originales. Además, cuando los argumentos de la función son pequeñas variables de tipo *int* o *double*, el detalle de duplicar la información no pasa de ser una mera anécdota. Pero si el argumento es un gran vector, una estructura que contiene matrices, etc., es fácil imaginar que el paso por valor puede afectar notablemente a la velocidad de ejecución y a la capacidad de memoria del ordenador. Por lo tanto, cuando se desea modificar los argumentos de la función, o cuando éstos sean muy grandes y el coste espacial y temporal de copiarlos sobre la pila sea muy grande, debe recurrirse a algún método alternativo.

Ahora surge la primera gran aplicación de los punteros, el paso por referencia de argumentos en las funciones. Para pasar parámetros por referencia, es decir, mediante la dirección y no el valor de las variables, hay dos posibilidades:

Paso por puntero

Lo que se pasa a la función es un puntero a los parámetros que se deben modificar. En el código de trabajo se ha definido la función *swapping2*, a la que se le pasan como argumento dos punteros de las variables a modificar :

```
void swapping2(int *i, int *j)   //paso por puntero (se duplica la dirección)
{
int temp=*i; *i=*j; *j =temp;
cout << "i=" << *i << "j=" << *j <<endl;
}
```

La llamada a esta función será:

```
case 2:                          //SI cambia el valor original de las variables
swapping2(ix , iy);              //x=2, y=1
```

En este caso sí que se intercambian los valores entre las variables. Para operar con los valores recurrimos al operador de indirección.

Paso por referencia

Para conseguir mayor claridad en el código y no tener que trabajar con el operador de indirección sobre el puntero, puede optarse por utilizar el paso por referencia de la variable. La llamada a la

función será análoga al paso por valor, pero en cambio sí que podrá modificarse el valor de las variables.

Se ha definido la función *swapping3*, a la que se le pasan como argumentos referencias de las variables a modificar:

```
void swapping3 (int &i, int &j)//paso por referencia.( Mezcla de 1 y 2)
{                               //solo se duplica la direccion. Codigo análogo al caso 1
int temp=i; i=j; j =temp;
cout << "i=" << i << "j=" << j <<endl;
}
```

La llamada a la función será:

```
default:  //SI cambia el valor original de las variables
swapping3(x,y);                 //x=2, y=1
```

Nótese que el código es similar a la función *swapping1*, pero en cambio los argumentos contienen el operador de referencia.

4 Ejercicios

4.1 Modificar swapping2 de manera que intercambien las direcciones, ¿qué pasa con las variables?

Solución al ejercicio.

```
// Ficha 12c
/* Modificar swapping 2 de manera que  intercambien
las direcciones. + Que pasa con las variables ? */

# include <iostream.h>

/*declaración de funciones para intercambio de las direcciones de las variables*/

void swapping2 (int *i, int *j);

int main (void)     //programa principal
{
//valores y direcciones iniciales  de las variables

int x=1, y=2;       //declaración e inicialización de variables, x=1, y=2
int *ix, *iy;       //declaración de punteros
ix=&x ; iy=&y;      //inicialización de punteros  &x=0X3562, &y =0X355E

cout <<"x=" << x <<" direccion x="<< &x << " y="<< y <<" direccion y="<< &y <<endl;

swapping2 (ix, iy);        //x=1, y=2    &x=0X3562, &y =0X355E
cout <<"x=" << x <<" direccion x="<< &x << " y="<< y <<" direccion y="<< &y <<endl;

return 0;
}

/*definición de funciones para intercambio de las direcciones de la variables*/

void swapping2(int *i, int *j)   // Paso por puntero
{                   //NO cambia el valor inicial de las variables
```

```
int *temp;          //SI cambia las direcciones temporales
temp=i; i=j; j =temp;
cout << "valor =" << *i << " en la direccion de i=" << i << endl;
cout << "valor =" << *j << " en la direccion de j=" << j << endl;
}
```

5 Código de trabajo

```
// Ficha 12c
/* Operadore new y delete */
# include <iostream.h>

class Matrix
{
public:
int size;
int *x;

 Matrix (int i): size (i) //constructor
//reserva de memoria dinámica
 {x = new int[size]; for (int j=0; j<i; j++)  {x[j]=0;}}
 //inicialización de la matriz en tamaño y número.
~ Matrix () { delete [] x; }     //destructor
//se elimina la memoria reservada para operar con la matriz
};

int main (void)
{
int size;
cout << "escribe el tamaño del array="<<endl;
cin >> size;
Matrix A (size);
return 0;
}
```

6 Conceptos

6.1 Operadores new y delete

El otro gran punto fuerte de los punteros es la gestión dinámica de memoria. Hasta ahora, para almacenar datos se ha usado la memoria estática. Al arrancar el programa se reserva toda la memoria necesaria y no se libera hasta que ha terminado el programa (int a[1000];).

Por supuesto, esta estrategia, cuanto más generalista sea el programa, tanto más ineficiente será; muchas veces no se sabe cuál es el tamaño de los vectores, por lo tanto es más lógico esperar que el programa solicite recursos a medida que los necesite y que posteriormente los libere cuando ya no se utilicen.

En C++ se dispone de mecanismos potentes para la gestión de la memoria dinámica, los operadores *new y delete.* El operador *new* sirve para asignar un espacio de memoria en tiempo de ejecución del programa, mientras que *delete* sirve para liberarlo.

Cuando se desee utilizar un vector de tamaño desconocido, será necesario definir un puntero y después asignarle un espacio de memoria mediante la instrucción *new*, por ejemplo:

```
int  *x = new  int  [1000];
```

De esta forma se ha creado un vector de 1000 enteros, siendo *x* el puntero que contiene la dirección del principio del bloque reservado. Posteriormente, para liberar la memoria ocupada se escribirá:

delete [ ] x; //en C++ 2.1 no es necesario decir cuántos enteros hay que borrar

Atención porque, si se hubiese escrito únicamente *delete x;* sólo se habría destruido el primer elemento del vector.

Como curiosidad nótese que para inicializar una variable se puede utilizar la siguiente sintaxis:

int *x = new int (10); //paréntesis , no corchetes

En este caso, se crea un entero con valor 10. Esto es, se le pasa el valor de inicialización como si se tratara de una función. Esta forma de inicializar se puede ver con mayor detalle en la ampliación de conceptos.

7 Ejemplo

7.1 Versión 1. Se inicializa de forma dinámica una matriz que en definitva es sólo un vector.

```
#include <iostream.h>
#include <stdlib.h>

int main()
{
 int *iMat; unsigned long size;
 cout << "escribe las dimensiones de la matriz "; cin >> size;

 iMat=new int [size];
 if (!iMat)
      {cerr << "Error. No hay memoria suficiente" << endl; exit(1);}

 cout << "entrada de datos " <<endl;
 for (unsigned long i=0; i<size; i++)
 {
  cout << " coeficiente (" << i << ")="; cin >> *iMat; iMat++;
 }

 iMat--;
 cout << "matriz ++++++++ " <<endl;
 for (i=0; i<size; i++)
 {
  cout << " " << *iMat--;
 }

 return 0;
}
```

7.2 Versión 2. Lo mismo, pero ahora la matriz en lenguaje natural se almacena en un vector. Obsérvese cómo los índices i,j varían para moverse a lo largo del vector.

```
#include <iostream.h>
#include <stdlib.h>

int main()
```

```
{
 double *s; unsigned long mfil,ncol,size;

 cout << "escribe las dimensiones de la matriz \nfilas="; cin >> mfil;
 cout << "columnas="; cin>>ncol; size=mfil*ncol;

 s=new double [size];
 if (!s)
       {cerr << "Error. No hay memoria suficiente" << endl; exit(1);}

 cout << "entrada de datos " <<endl;
 for (unsigned long i=1; i<=mfil; i++)
       for (unsigned long j=1; j<=ncol; j++)
        {
       cout << " coeficiente (" << i <<"," << j << ")=";
        cin >> s[ncol*(i-1) + j - 1];
       }

 double *dMat=&s[0];
 cout << "matriz ++++++++ " <<endl;
 for (i=0; i<mfil; i++)
       { cout << endl;
              for (unsigned long j=0; j<ncol; j++)
                     cout << " " << *dMat++;
       }
 return 0;
}
```

7.3 Versión 3. Se introduce el concepto de clase para matrices visto en fichas anteriores y se trabaja con el dimensionamiento dinámico de los datos. Atención también al operador () para introducir y extraer coeficientes de la matriz.

```
#include <iostream.h>
#include <stdlib.h>

typedef unsigned long ilong;

class Matrix
{
 public:
  unsigned long nr, nc, size;
  double*    s;

    // constructors
    Matrix () : nr(0), nc(0), s(0), size(0) {}
    Matrix (ilong m, ilong n): nr(m), nc(n), size(m*n)
     {
     s=new double[size];
     if (!s) {cerr << "no hay memoria" << endl; exit (1);}
        for (ilong i=0; i<size; i++) s[i]=0.;
        }

       //destructor
    ~Matrix() {delete [] s; s=0;}

       double& operator () (ilong m, ilong n)
       {
       if (m > nr || n > nc || m < 1 || n < 1)
              { cerr <<"indice fuera de las dimensiones. Error"; exit (1);}
       return s[nc*(m-1) + n-1];
       }

       friend ostream& operator << (ostream& s, const Matrix& m);
```

```
};


int main ()
{
 Matrix A(2,2);
 for (int i=1; i<=2; i++)
        for (int j=1; j<=2; j++)
        {cout << "coeficiente (" <<i <<"," << j<<")="; cin >> A(i,j);}

 cout <<A;
 return 1;
}


ostream& operator << (ostream& os, const Matrix& m)
{
 os << "Matrix ++++ "; if (!m.nr && !m.nc) os<<"void";
 for (ilong i=1; i<=m.nr; i++)
 {
  os <<endl;
  for (ilong j=1; j<=m.nc; j++)
        os << m.s[m.nc*(i-1) + j-1] <<" " ;
  }
 os << "\n";
 return os;
}
```

8 Ampliación de conceptos

8.1 Punteros a funciones

Hasta ahora se ha visto que las funciones se llamaban por su nombre; sin embargo, también se pueden utilizar desde un puntero. En C++ se puede tomar la dirección de una función para operar con ella, en lugar de operar con la función directamente. En este caso, un puntero a una función se define como:

tipo_ Retorno (*Puntero_Función) (tipo_Arg1, tipo_Arg2, ..., tipo_ArgN) ;

Obsérvese que si se sustituye (*Puntero-Funcion) por f se tendría la declaración de una función.

Algunos ejemplos de definiciones de punteros a funciones serían:

```
int     (*p)  ( );           /*puntero p a una función que retorna un entero y no tiene
                             argumentos */

void  (*p)  (int, int)       /* puntero p a una función que no retorna nada y tiene dos
                             argumentos enteros */
```

en cambio:

```
int      *f  ( );            /* atención, función f  que retorna un puntero a un entero */
```

Por lo tanto, para llamar a una función mediante un puntero se debe seguir la sintaxis de las dos primeras definiciones. Como el operador de llamada de función ( ) tiene mayor precedencia que el operador de indirección *, no se puede simplemente escribir *p ( ).

La utilidad de los punteros a funciones es limitada; su uso se justifica en aquellos casos en que es necesario utilizar varias funciones que hagan lo mismo de forma distinta y con los mismos argumentos. Por ejemplo, sería el caso de funciones de ordenación; los argumentos siempre son los mismos, pero el algoritmo interno puede ser diferente, entonces es más cómodo llamar a la función desde un apuntador que escribir cada función con un nombre distinto. Otro caso parecido sería las funciones de salida de resultados, donde se puede desear tener varios formatos, pero una única función salida.

Es preciso declarar los tipos de los argumentos de los punteros a funciones, tal como se hace con las funciones mismas. En las asignaciones de punteros debe haber una concordancia exacta del tipo completo de la función. Por ejemplo:

```
void (*pf) (char*);      // puntero pf a función del tipo void (char*)
void  f1   (char*);      // función f1 del tipo void (char*)
int   f2   (char*);      // función f2 del tipo int (char*)
void  f3   (int*);       // función f3 del tipo void (int*)

void f ( )
{
pf = &f1        ;        //correcto
pf = &f2        ;        //error de tipo devuelto
pf = &f3        ;        //error del tipo de argumento

(*pf) ("asdf");  //correcto
(*pf) (1);       //error de tipo de argumento

int i = (*pf) ("qwer");   // error void asignado a int
}
```

Las reglas de paso de argumentos son idénticas para las llamadas directas a una función y para las llamadas a una función a través de un puntero.

8.2 Puntero implícito this

Toda persona es uno mismo y puede hablar sobre sí mismo o cuidar su figura. De igual forma, toda clase puede hacer referencia a sus propios miembros. En general, el acceso a los miembros se realiza llamando a los métodos o modificando los datos. Sin embargo, el puntero *this* permite acceder a los miembros de una clase con la estrategia de los apuntadores.

El puntero *this* hace referencia a la propia clase, es un apuntador a ella misma y está siempre implícitamente definido, es decir, siempre está disponible y no es necesario inicializarlo o definirlo. Sabiendo que se puede acceder a los miembros de las clases llamándolos por su nombre, ¿dónde radica el interés de *this*? Bien, en general, el apuntador *this* no se utiliza demasiado a menos que se deba devolver la propia clase en un método o bien acceder a un miembro desde un apuntador.

En general, podemos dejar de usar *this* para direccionar los atributos de una clase. Es simplemente una forma didáctica e ilustrativa de mostrar que hay un parámetro oculto que indica a las funciones sobre qué objeto se está accediendo. A veces es necesario su uso para pasar un puntero al objeto actual desde un miembro, pero, en general, su uso es limitado. El puntero *this* tiene sus restricciones:

- no se pueden asignar valores a *this*.
- sí se pueden asignar valores a * *this*. Por ejemplo para copiar un objeto con otro.
- sí se puede retornar * *this* por referencia y por valor.

Por tanto, cuando un objeto llama a una función de una clase, esta función sabe qué objeto la está llamando gracias al primer parámetro, el parámetro implícito *this*. Este puntero hace a su vez que los atributos accedidos sean los del objeto en cuestión, no los de otro objeto de la misma clase.

Ficha 13: Herencia II y polimorfismo II. Clase abstractas

1 Objetivos generales

- Conceptos avanzados de la herencia y polimorfismo: conversiones implícitas en la herencia, herencia múltiple, funciones virtuales y clases abstractas.

2 Código de trabajo (conversiones implícitas en la herencia)

```
// Ficha 13
/* Conversiones implicitas en la herencia */

# include <iostream.h>

class cuadrilatero
{
public:
      int a,b;
      cuadrilatero (int lado1, int lado2) { a=lado1; b=lado2;}
      ~cuadrilatero () {}
};

class cuadrado : public cuadrilatero
{
public:
      cuadrado (int lado) : cuadrilatero (a,a) {}
      ~cuadrado () {}
};

int main (void)
{
 cuadrilatero b (1,2);
 cuadrado d(1);

b=d;
// d=b;
// d=(cuadrado) b;
// d=cuadrado (b);

cuadrado &rd = d;
cuadrilatero &rb=rd;
// cuadrado &rd2=b;
cuadrado &rd3= (cuadrado &)b;
```

```
027    cuadrilatero *pb;
028    cuadrado *pd;
029    pb=pd;
030    // pd=pb;
031    pd=(cuadrado *) pb;

032    return 0;
033    }
```

3 Conceptos

3.1 Conversiones implícitas en la herencia

Están definidas cuatro conversiones implícitas que se aplican entre una clase derivada y su clase base:

1. Un objeto de la clase derivada será siempre implicitamente convertido a un objeto de la clase base, tal y como se señala en la línea **019**. Lo contrario no es posible, líneas **020, 021** y **022**. Píensese en un objeto derivado como el base y algo más, esto significa que el derivado podrá trabajar como un base, pero un base nunca podrá hacer todo lo que hace el derivado. Por ello la conversión implícita sólo se realiza en la línea **019**. Únicamente si se definiera un constructor de conversión o un operador de conversión, se solventarían los problemas de las tres líneas siguientes **020-022**.

2. Una referencia a una clase derivada será implícitamente convertida a una referencia a la clase base, línea **024**. Al revés, tal y como aparece en la línea **025**, se tendrá que hacer una conversión explícita, línea **026**, para que pueda ser admitido por el compilador.

3. Un puntero a una clase derivada será implícitamente convertido a una clase base (línea **029**). Al revés se cometerá un error según la línea **030**; para evitarlo se dederá hacer una conversión explícita, como en la línea **031**.

4. Un puntero a un miembro de la clase base será convertido implícitamente a un puntero a miembro de la clase derivada.

4 Código de trabajo (herencia múltiple)

```
// Ficha 13
/* herencia múltiple.Conflictos en duplicidad información*/

# include <iostream.h>

class circulo;              // clase ya definida
class circunferencia;       // clase ya definida
class topo : public circunferencia , circulo  //herencia multiple.
                                       //lista de derivación
{
topo (int center, int radius) : circunferencia (center,radius),
                    circulo (center, radius){}
// !!!!! duplicidad de variable comunes
};

int main (void)
{
topo  Red_one  (center ,  radius);//objeto Red_one de clase topo
topo. perimetro (); //función perímetro de la clase topo
return 0;
}
```

5 Conceptos

5.1 Herencia múltiple. Conflictos en la duplicidad de información y en la compartición de métodos

Cuando una clase necesita heredar más de una clase se recurre a la *herencia múltiple*. Este mecanismo intenta combinar distintas características de las diferentes clases para formar una nueva clase diferenciada que cumpla todas las características de sus clases bases.

En el código de trabajo, en la línea **106**, se ha definido la clase *topo* como derivada *public* de dos clases base, la clase *circumferencia* y la clase *circulo.* El constructor de la clase *topo* (línea **108**) utiliza los constructores de las clases bases de las cuales deriva y son llamados en el siguiente orden:

1º) Los constructores de la clase base en el orden, en que se han declarado en la lista de derivación (la lista de derivación es el conjunto de clases que se derivan y que van a continuación del nombre de la clase); *circunferencia (center, radius)* y *circulo (center, radius).*

2º) El constructor de la clase derivada; *topo (center, radius)*

Los destructores son llamados en orden inverso.

Obsérvese que siempre que haya dos o más miembros de dos o más clases bases, directa o indirectamente, con el mismo identificador, se produce una ambiguedad y el compilador señala error, aunque un identificador sea privado y el otro público.

En la clase *topo* definida en el código de trabajo, los datos miembros son dos números enteros; uno para especificar el centro (*center*) y otro para especificar el radio (*radius*) de las clases base de las cuales deriva (*circunferencia* y *circulo*), las cuales también utilizan el mismo identificador para sus datos miembros. Esto hace que haya una duplicidad de información al definir (línea **113**) un objeto (*Red_one*) de la clase derivada (*topo*). Por otra parte, si una función miembro (*perimetro)* utiliza el mismo identificador en las dos clase bases, al llamar a esta función (línea **114**) el compilador señala error porque no sabe a qué clase se refiere. En consecuencia hay que redefinir el miembro en la clase derivada o explicitar con el operador de campo :: desde que clase se está llamando a dicho miembro.

6 Código de trabajo (funciones virtuales)

```
// Ficha 13
/* Polimorfismo entre clase heredadas. Funciones virtuales */

# include <iostream.h>

class rectangulo
{
public:
   int a,b;
   rectangulo (int lado1, int lado2) { a=lado1; b=lado2;}
   virtual ~rectangulo () {}
   virtual int Area(void)
         {cout << "estoy en rectangulo" << endl; return a*b;}
};

class cuadrado : public rectangulo
{
public:
```

```
216    cuadrado (int lado) : rectangulo (lado,lado) {}
217    ~cuadrado () {}
218    int Area(void) {cout << "estoy en cuadrado" << endl; return a*a;}
219 };


220 int main (void)
221 {
222    rectangulo r1 (1,1); cuadrado c1 (2);
223    cout << "area rectangulo " << r1.Area() << endl;
224    cout << "area cuadrado " << c1.Area() << endl;
225    rectangulo *p1; p1=&c1;
226    cout << "area" << p1->Area() << endl;
227    return 0;
228 }
```

7 Conceptos

7.1 Funciones virtuales. Polimorfismo entre clases heredadas

Recordemos que el *polimorfismo* es la habilidad de un método de modificar su implementación en función del tipo de objeto que lo llama. El C++ hace uso de las *funciones virtuales* para emplear el polimorfismo entre clases heredadas. Supóngase una clase base con una función llamada *Area* (línea **210**) y una clase derivada que redefine la función (línea **218**), en tiempo de ejecución y dado que ambas clases, base y derivada, son accesibles ¿cómo puede saber el programa si llama a la función de la base o de la derivada, cuando se utiliza un apuntador del tipo base?

En el código de trabajo se han definido dos clases (*rectangulo* y *cuadrado*) (líneas **204** y **213**) y dos instanciaciones de las clases, es decir, dos objetos en la línea **222**.

El programa llama a un rectángulo y a un cuadrado y calcula sus áreas correctamente en las líneas **223** y **224**. Esto es debido a que el compilador comprueba el tipo de la variable y, dependiendo de cuál sea, llama a la función correspondiente. Esto se conoce como *enlace estático* (*static binding*) y significa que el enlace entre objeto y método es estático (en el momento de compilación).

Sin embargo, en la línea **225** se define un apuntador de tipo clase base y recibe la dirección de la clase derivada; esto es posible por la conversión explícita de la herencia, pero al llamar a la función *Area*, realmente ¿a cuál se está llamando? Para evitar problemas de este tipo, se recurre al denominado *enlace dinámico* (*dynamic binding*), que consiste en que el método llamado no depende del tipo de variable que usemos para referenciarlo, sino que dependerá de la clase del objeto. Por lo tanto aunque el apuntador tipo es de la base, el objeto apuntado es derivado y en consecuencia calcula el área de un cuadrado.

En funciones que vayan a tener un puntero para ligar el objeto con su método definiremos la función como *virtual* (línea **210**), es decir, como una función miembro especial que se llama a través de una referencia o puntero a la clase base y que se enlaza dinámicamente en tiempo de ejecución.

Ahora ya se puede hablar de verdadero polimorfismo. Se puede trabajar con punteros o referencias a *rectangulo* (algunos de los cuales pueden ser realmente de tipo *rectangulo* y otros *cuadrado*) con la seguridad de que se llamarán a las funciones correctas.

Hay que destacar que el enlace dinámico de funciones *virtuales* sólo es posible con punteros o referencias a clases bases.

En general hay cinco causas para que se pierda la *virtualidad*:

1. Cuando una función virtual es invocada mediante un objeto (y no una referencia o puntero).
2. Cuando se pasan parámetros por valor.
3. Cuando, siendo una referencia o puntero, se especifica usando el operador de campo :: .
4. Cuando una función virtual es invocada dentro del constructor de la clase base.
5. Cuando una función virtual es invocada dentro del destructor de la clase base.

7.2 Destructores virtuales

Los destructores se pueden declarar virtuales. En las líneas de código **204** y **213** se ha definido una clase *cuadrado* derivada de *rectangulo*.

Se han definido dos objetos; uno de tipo *cuadrado* y otro de tipo *rectangulo*, así como un puntero de tipo *rectangulo* (línea **225**), el cual apunta realmente a objeto *cuadrado* (línea **225**). La destrucción del *cuadrado* a través del apuntador pondría en aprietos al sistema, por aquello de cuál es el destructor llamado. Para evitarlo se define el destructor de la clase base como virtual, entonces se producirá el polimorfismo entre clases heredadas tal y como sucede con los métodos normales. De esta forma, se llamaría al destructor correspondiente al objeto, al hacer *delete p1*.

Otra forma de resolver el problema sería mediante un *cast* del apuntador, pero perdería toda la elegancia.

Hay que destacar que los constructores no pueden ser virtuales, porque necesitan información acerca del tipo exacto de objeto que se va a crear.

8 Código de trabajo (clases abstractas)

```
// ficha 13
/* Clases abastractas */

# include <iostream.h>

class poligono
{
 public:
    poligono (void) {}
    ~poligono () {}
    virtual double Area (void) =0;
};

class rectangulo : public poligono
{
 public:
   int a,b;
   rectangulo (int lado1, int lado2): poligono ()
          { a=lado1; b=lado2;}
   virtual ~rectangulo () {}
   double Area(void)
          {cout << "estoy en rectangulo" << endl; return a*b;}
};

class cuadrado : public rectangulo
{
public:
   cuadrado (int lado) : rectangulo (lado,lado) {}
   ~cuadrado () {}
```

```
326    double Area(void)
327           {cout << "estoy en cuadrado" << endl; return a*a;}
328 };

329 class triangulo : public poligono
330 {
331  public:
332   double b, h;
333   triangulo (double base, double altura) : poligono ()
334           {b=base; h=altura;}
335   ~triangulo () {}
336   virtual double Area(void)
337           {cout << "estoy en triangulo" << endl; return b*h/2.;}
338 };

339 int main (void)
340 {
341  rectangulo r1 (1,1); cuadrado c1 (2); triangulo t1(2.,3.);
342  cout << "area rectangulo " << r1.Area() << endl;
343  cout << "area cuadrado " << c1.Area() << endl;
344  cout << "area triangulo " << t1.Area() << endl;
345  poligono * p1;
346  p1=&r1;
347  cout << p1->Area() << endl;
348  return 0;
349 }
```

9 Conceptos

9.1 Clases abstractas

Una clase *abstracta* (línea **304**) es aquella que sólo sirve como base o patrón para otras clases (líneas **311, 321** y **329**) y no puede ser instanciada, es decir, no pueden haber objetos de esa clase. Nótese que dentro del *main* no hay ningún objeto de tipo *poligono*.

Una clase se convierte en *abstracta* cuando tiene alguna *función virtual pura* dentro de su definición. La sintaxis de este tipo de funciones es:

virtual TipoRetorno NombreFunción (Tipo1 a, Tipo 2 b) = 0 ;

como en la línea **309**. Al igualar a cero la función *virtual* significa que el método se deja sin definir y que alguna clase heredada se encargará de precisar para qué sirve.

Una clase *abstracta* no se puede instanciar porque no tiene sentido que un objeto pueda llamar a funciones sin definir que, por lo tanto, no pueden usarse. En cambio, sí que se pueden definir punteros y referencias de tipo abstracto (línea **345**) porque a través del enlace dinámico las clases derivadas se encargarán de definir la función. Esto último sucede en la línea **347**, donde el apuntador tipo *poligono* llama a la función *Area()* desde la clase derivada a la que apunta (un objeto de tipo *rectangulo*).

Si no se pueden crear objetos de tipo de abstracto, ¿cuál es el sentido de dichas clases? Bueno, las clases abstractas pretenden ayudar a crear una programación con sentido, las clases abstractas son patrones que contienen todos los métodos que van a heredar sus clases derivadas, por ello sirven para crear el marco general de un desarrollo. Atención porque cuando se habla de patrones no hay que confundir con *templates* que es un concepto diferente que aparece en la ficha siguiente.

Las clases *abstractas* sirven como base de muchas jerarquías. Construir una jerarquía congruente suele ser difícil al principio, pero si está bien organizada cualquier modificación posterior es heredada, con lo que sólo se tiene que retocar una vez. Esta es la ventaja de un programa orientado a objetos.

Las clases heredadas de una clase *abstracta* deben declarar todos los métodos puros heredados (líneas **318**, **326** y **336**). En el caso de que no se redefina o se redefina también puro, la clase heredada también será *abstracta*. Hay que destacar que está prohibido que una clase heredada declare como puro un método que en la clase base no lo es. Esto es así porque entonces se perdería el sentido de la abstracción, que siempre va de la madre hacia las hijas y no al revés.

10 Ejercicios

10.1 Definir una clase abstracta para figuras geométricas que permita el cálculo de áreas y salida de resultados.

Solución al ejercicio

```
// Ficha 13
/* Definir una clase abstracta para figuras geométricas
que permita el cálculo de áreas y salida de resultados */

# include <iostream.h>

class abstracta
{
 public:
       abstracta (void){}
       virtual double area () = 0;
       virtual double volumen () =0;
};


class poliedro : public abstracta
{
public:
       poliedro () : abstracta () {};
       ~poliedro () {}
       double area () { cerr << "area sin sentido "; return 0.;}
       double volumen () =0;
};

class cubo : public poliedro
{
public:
 double c;
 cubo (double lado) : poliedro () {c=lado;}
 ~cubo () {}
 double volumen(void) {cout << "volumen del cubo ="; return c*c*c;}
};


class poligono : public abstracta
{
 public:
        poligono (void) : abstracta () {}
        ~poligono () {}
        double area () =0;
        double volumen () { cerr << "volumen sin sentido "; return 0.;}
};
```

```
class rectangulo : public poligono
{
public:
       double a,b;
       rectangulo (double lado1, double lado2): poligono () { a=lado1; b=lado2;}
       virtual ~rectangulo () {}
       double area(void) {cout << "area del rectangulo ="; return a*b;}
};


class cuadrado : public rectangulo
{
public:
       cuadrado (double lado) : rectangulo (lado,lado) {}
       ~cuadrado () {}
       double area(void) {cout << "area del cuadrado ="; return a*a;}
};


int main (void)
{
 rectangulo r1 (1.,1.); cuadrado c1 (2.);
 cubo q1(2.);

 cout << r1.area() << endl;
 cout <<  c1.area() << endl;
 cout << q1.volumen() << endl;

 // abstracta a1(); error de compilación no es instanciable
 abstracta * a1; a1=&r1;
 cout << a1->area() << endl;
 cout << a1->volumen() << endl;

 // poligono p1(); error de compilación no es instanciable
 // poligono * p1; p1=&q1; tipos incompatibles
 poligono * p1; p1=&c1;

 return 0;
}
```

11 Ejemplo

11.1 Versión 1. Se define la clase para matrices con la definición de los constructores y la redefinición de operadores como la suma con la intención de aproximar la sintaxis al pensamiento natural. La operación suma se ha planteado ahora de forma alternativa, para que se pueda ver que hay muchas soluciones para un mismo problema. Obsérvese el código del programa principal, ¿ a que se entiende todo?

```
#include <iostream.h>
#include <stdlib.h>

class Matrix
{
 public:
  unsigned long nr, nc, size;
  double*  s;

    // constructors
    Matrix () : nr(0), nc(0), s(0), size(0) {}
    Matrix (unsigned long m, unsigned long n)
```

```
      {
       if(m<0 || n<0)
       cerr << " dimension imposible, tomo valor absoluto" << endl;
       nr=abs(m); nc=abs(n); size=nc*nr;
       s=new double[size];
       if (!s) { cerr << "no hay memoria" << endl; exit(1);}
        for (unsigned long i=0; i<size; i++) s[i]=0.;
       }

       // copy constructor
       Matrix (const Matrix& m);

       //destructor
     ~Matrix() {delete [] s; s=0;}

       // acceso a datos
     double& operator () (unsigned long m, unsigned long n);
     Matrix& operator = (const Matrix& m); // asignacion

       // operadores aritmeticos
       Matrix& operator += (const Matrix& m);
       friend Matrix operator + (const Matrix& a, const Matrix &b);
       // salida de resultados
       friend ostream& operator << (ostream& os, const Matrix& m);

};


//--- Matrix operator ()
double& Matrix::operator() (unsigned long m, unsigned long n)
{
  if (m > nr || n > nc || m < 1 || n < 1)
  { cerr << "indices fuera de limites " <<endl; exit(1);}
  return s[nc*(m-1) + n-1];
}


int main()
{
 Matrix A(2,2);
 for (int i=1; i<=2; i++)
  for (int j=1; j<=2; j++)
  {cout << "coeficiente (" <<i <<"," << j<<")="; cin >> A(i,j);}

 Matrix B(A); Matrix C=A+B;
 cout << A << B << C ;

 return 0;
}


//copy constructor
Matrix::Matrix(const Matrix& m) : nr(m.nr), nc(m.nc), size(m.nr*m.nc)
{
  s=new double[size];
  if (!s) { cerr << "no hay memoria" << endl; exit(1);}
  for (unsigned long i=0; i<size; i++) s[i]=m.s[i];
}

//--- operator =
Matrix& Matrix::operator = (const Matrix& m)
{
  if(this == &m) return *this;
  if (size != m.size)
    {
```

```
      delete [] s;
      s=new double[size];
         if (!s) { cerr << "no hay memoria" << endl; exit(1);}
    }
  nr = m.nr;
  nc = m.nc;
  size = m.size;
  for (unsigned long i=0; i<size; i++) s[i]=m.s[i];
  return *this;
}


//--- operator +=
Matrix& Matrix::operator += (const Matrix& m)
{
 if (size == 0 && m.size != 0) return *this = m;
 if ((size == 0 && m.size == 0) || (size!=m.size))
       { cerr << "sumando matrices de dimensiones erroneas" << endl; exit(1);}

 for (unsigned long i=0; i<size; i++) s[i]+=m.s[i];

 return *this;
}

//--- operator +
Matrix operator + (const Matrix& a, const Matrix &b)
{

 if ((a.size == 0 && b.size == 0) || (a.size!=b.size))
       { cerr << "sumando matrices de dimensiones erroneas" << endl; exit(1);}

 Matrix sum=a;
 sum+=b;
 return sum;
}

// salida de datos
ostream& operator << (ostream& os, const Matrix& m)
{
 os << "Matrix ++++ "; if (!m.nr && !m.nc) os<<"void";
 for (unsigned long i=1; i<=m.nr; i++)
 {
  os <<endl;
  for (unsigned long j=1; j<=m.nc; j++)
        os << m.s[m.nc*(i-1) + j-1] <<" " ;
  }
 os << "\n";
 return os;

}
```

11.2 Versión 2. En este caso se utiliza la herencia para trabajr con matrices y vectores de forma parecida a la ficha 10. Observar la redefinición de operadores y el linkado en tiempo de ejecución de las funciones polimórficas.

```
#include <iostream.h>
#include <stdlib.h>

class Matrix
{
 public:
  unsigned long nr, nc, size;
  double*  s;
```

```
    // constructors
    Matrix () : nr(0), nc(0), s(0), size(0) {}
    Matrix (unsigned long m, unsigned long n)
     {
       if(m<0 || n<0)
        cerr << " dimension imposible, tomo valor absoluto" << endl;
       nr=abs(m); nc=abs(n); size=nc*nr;
       s=new double[size];
       if (!s) { cerr << "no hay memoria" << endl; exit(1);}
       for (unsigned long i=0; i<size; i++) s[i]=0.;
        }

       // copy constructor
       Matrix (const Matrix& m);

       //destructor
    virtual ~Matrix() {delete [] s; s=0;}

       // acceso a datos
    double& operator () (unsigned long m, unsigned long n);
       Matrix& operator = (const Matrix& m); // asignacion

       // operadores aritmeticos
       Matrix& operator += (const Matrix& m);
       friend Matrix operator + (const Matrix& a, const Matrix &b);
       friend ostream& operator << (ostream& os, const Matrix& m);

};


//--- Matrix operator ()
double& Matrix::operator() (unsigned long m, unsigned long n)
{
  if (m > nr || n > nc || m < 1 || n < 1)
  { cerr << "indices fuera de limites " <<endl; exit(1);}
  return s[nc*(m-1) + n-1];
}

class Vector : public Matrix{
  public:
       Vector () : Matrix () {}
       Vector(unsigned long l) : Matrix(l,1) {}
       Vector(const Matrix& m) : Matrix(m) {}

    double& operator () (unsigned long m);
       friend ostream& operator << (ostream& s, const Vector& m);

};


double& Vector::operator () (unsigned long m)
  {
       if (m > nr || m < 1)
       { cerr << "indices fuera de limites " <<endl; exit(1);}
       return s[m-1];
  }


int main()
{
 Matrix A(2,1);
 for (int i=1; i<=2; i++)
  for (int j=1; j<=1; j++)
  {cout << "coeficiente (" <<i <<"," << j<<")="; cin >> A(i,j);}
```

```
 Vector B(A);

 Vector D=A+B;
 Matrix C=A+B;
 cout << A << B << C << D;

 return 0;
}


//copy constructor
Matrix::Matrix(const Matrix& m) : nr(m.nr), nc(m.nc), size(m.nr*m.nc)
{
  s=new double[size];
  if (!s) { cerr << "no hay memoria" << endl; exit(1);}
  for (unsigned long i=0; i<size; i++) s[i]=m.s[i];
}

//--- operator =

Matrix& Matrix::operator = (const Matrix& m)
{
  if(this == &m) return *this;
  if (size != m.size)
    {
      delete [] s;
      s=new double[size];
         if (!s) { cerr << "no hay memoria" << endl; exit(1);}
    }
  nr = m.nr;
  nc = m.nc;
  size = m.size;
  for (unsigned long i=0; i<size; i++) s[i]=m.s[i];
  return *this;
}


//--- operator +=

Matrix& Matrix::operator += (const Matrix& m)
{
 if (size == 0 && m.size != 0) return *this = m;
 if ((size == 0 && m.size == 0) || (size!=m.size))
       { cerr << "sumando matrices de dimensiones erroneas" << endl; exit(1);}

 for (unsigned long i=0; i<size; i++) s[i]+=m.s[i];

 return *this;
}

//--- operator +

Matrix operator + (const Matrix& a, const Matrix &b)
{

 if ((a.size == 0 && b.size == 0) || (a.size!=b.size))
       { cerr << "sumando matrices de dimensiones erroneas" << endl; exit(1);}

 Matrix sum=a;
 sum+=b;
 return sum;
}
```

```
//--- salida de datos
ostream& operator << (ostream& os, const Matrix& m)
{
 os << "Matrix ++++ "; if (!m.nr && !m.nc) os<<"void";
 for (unsigned long i=1; i<=m.nr; i++)
 {
  os <<endl;
  for (unsigned long j=1; j<=m.nc; j++)
        os << m.s[m.nc*(i-1) + j-1] <<" " ;
  }
 os << "\n";
 return os;

}

//--- salida de datos
ostream& operator << (ostream& os, const Vector& m)
{
 os << "Vector ++++ "; if (!m.nr && !m.nc) os<<"void";
 for (unsigned long i=1; i<=m.nr; i++)
 {
  os <<endl;
  for (unsigned long j=1; j<=m.nc; j++)
        os << m.s[m.nc*(i-1) + j-1] <<" " ;
  }
 os << "\n";
 return os;
}
```

12 Ampliación de conceptos

12.1 Herencia virtual

En una jerarquía de clases heredadas puede ocurrir que una clase se pueda heredar dos veces por dos caminos diferentes (*herencia duplicada*). Cuando esto ocurre pueden aparecer ambigüedades ya que se puede llegar a tener dos copias de un mismo miembro. En efecto, en el ejemplo siguiente:

class A
{ public:
int a; };

class B : public A
{ public:
int b; };

class C: public A
{ public:
int c; };

class D : public B, public C
{ public:
int d; };

la clase D tendrá una copia de entero *d* , una copia de *b*, una copia de *c*, pero dos copias del entero *a*; una que proviene de la herencia de B y otra que proviene de la de C.

Por tanto, al definir un objeto de la clase D y acceder a su miembro *a*, se producirá una ambiguedad:

```
D obj;
obj.a = 0;          //error
```

La solución sería indicar a qué clase pertenece *a* mediante el operador de acceso **::**

```
obj. B :: a = 0;
```

Cuando sólo se desea una copia de *a* y no dos, se recurre a la *herencia virtual*, que consiste en heredar virtualmente la clase A dentro de B y C, añadiendo el identificador *virtual* antes de la clase base.

```
class B : virtual public A {..};
class C : virtual public A {..};
```

Se pueden mezclar herencias virtuales y normales, y el resultado es que todas las herencias virtuales formaran una sola copia mientras que cada herencia normal formará una propia. Ejemplo:

```
class B : virtual public A {..};
class C :  public A {..};
class D : virtual public A {..};
class E :  public A {..};
class F :  public B, public C, public D, public E {..};
```

La clase F tiene 3 copias de A; una por parte de C, otra por parte de E y otra por parte de todas las herencias virtuales (B y D). Se recomienda heredar siempre virtualmente por los problemas que pudieran aparecer mas tarde al ampliar las jerarquías.

Ficha 14: Patrones (templates)

1 Objetivos generales

- Conocer qué son los *templates* y cómo se pueden utilizar.

2 Código de trabajo

```
// Ficha 14
/*patrones*/

# include <iostream.h>

template <class T> T Maximus (T a,T b)
{
 if (a>b) return a;
 else return b;
  }

int  main (void)
{
  int ia=5,ib=10;
  double da=3.0,db=5.1;

  int ic = Maximus (ia,ib);
  double dc = Maximus (da,db);
   cout << "el maximo entero es " << ic << endl;
   cout << "el maximo real es " << dc << endl;
    return 0;
}
```

3 Conceptos

3.1 Uso de templates

El concepto de *template* se puede traducir por plantilla o patrón. Los *templates* permiten definir clases y funciones patrón sin especificar el tipo de dato o, en otras palabras, sirven para distintos tipos de datos. El *template* es útil cuando diferentes tipos de datos han de ser manejados de igual forma. Por ejemplo, se pueden tener enteros, reales, cadenas, incluso objetos más complicados, y desear ordenarlos bajo un cierto criterio, de menor a mayor o viceversa, como sería el caso de la función

Maximus del código de trabajo (línea **004**). En esta ocasión, se debería definir una función que ordenara cada tipo de dato, lo cual aumenta negativamente el volumen de código. La solución se encuentra en los *templates*; basta con definir una función patrón que ordene *cualquier* tipo de dato. Justamente la función de la línea **004** donde el parámetro *T* representa cualquier tipo de dato existente y concocido.

3.2 Funciones patrón

El prototipo de una función patrón es:
1. La palabra *template*.
2. Una cláusula, entre los símbolos menor y mayor, donde se incluye el *tipo* de datos precedido de la palabra *class*. Aunque se antepone la palabra *class* al tipo, se refiere a cualquier tipo, sea clase o no, como pueden ser los enteros, reales, etc.
3. La definición de una función cuyos argumentos deben ser, como mínimo una vez, cada uno de los tipos de la lista *template*.

Una función patrón puede entenderse, por tanto, como un conjunto ilimitado de funciones sobrecargadas, pudiéndose instanciar todos los tipos compatibles con la definición.

En el código de trabajo se ha definido una función para calcular el máximo de dos valores de *tipo genérico T*. En la línea **004** se define una *plantilla* para un *tipo genérico T*. Nótese que la función recibe dos argumentos de tipo *T* (T a, T b) y devuelve un argumento tipo *T* , basta con ver el *return* de las líneas **006** y **007**. En las líneas dentro del *main*, **013** y **014**, se llama dos veces a la función *Maximus* para inicializar dos variables de distinto tipo *ic* y *dc*. En particular, en la línea **013** se realiza una *instanciación* del *tipo Max < int , int >,* mientras que en la línea **014** es del tipo *Max < double , double >*.

La alternativa a esta estrategia de programación pasa por escribir dos funciones diferentes que reciben argumentos y devuelven tipos particulares, es decir:

int Maximus (int a, int b);
double Maximus (double a, double b);

Lógicamente, para que la comparación de valores funcione deben estar definidos los operadores < y > para los tipos susceptibles de utilizar la función *template.* En el ejemplo, estos operadores vienen por defecto en el lenguaje, ya que son tipo *int* y tipo *double,* en caso contrario se deben definir en algún sitio los operadores sobre los tipos pertinentes. ¿Qué pasaría si deseamos comparar complejos? No funcionaría el programa hasta que hubieramos definido la clase complejo y las operaciones de comparación.

4 Código de trabajo

```
// Ficha 14
/*patrones*/

# include <iostream.h>

template <class T> class MyArray
{
 public:
T* datos; int length;
```

```
108    MyArray (int size)
109    {
110     datos = new T[size]; length=size;
111     cout << endl << "definendo nuevos " << size << " datos" << endl;
112     for (int i=0; i<length; i++)
113     { cout << " escribe dato " << i << " "; cin>>datos[i];}
114    }

115    ~MyArray (void) { delete [] datos; datos=NULL;}

116    void PrintIt(void)
117    {
118     for (int i=0; i<length; i++) cout << datos[length-i-1] <<endl;
119    }
120    };

                    // final de la clase MyArray

122     int  main (void)
123    {
124    MyArray <int> Loteria(3);
125    Loteria.PrintIt();
126    MyArray <double> TimeRecords(4);
127    TimeRecords.PrintIt();
128    return 0;
129    }
```

5 Conceptos

5.1 Clases patrón

El mismo concepto que se había visto para las funciones *template* es extensible a las clases. Es decir, se pueden definir clases *template* que sirvan para almacenar diferentes tipos de datos que deban operarse con los mismos métodos.

En el código de trabajo se presenta una clase *template MyArray*, línea **104**. Esta clase pretende guardar un vector de diferentes entidades, por ejemplo *int* y *double* como aparece en el código principal, líneas **124** y **126**. El único método que contiene es una impresión *PrintIt* en orden inverso, líneas **116-119**. Obsérvese que ambos vectores, *Loteria* (por ejemplo guarda los últimos números ganadores) y *TimeRecords* (por ejemplo guarda los últimos records de una carrera), sirven para guardar diferentes tipos de datos que se agrupan bajo el apuntador *T* datos*, línea **107**. Se supone que pueden ser de diferente tipo, aquel que el usuario necesite, pero siempre se va a tratar de la misma forma, imprimir en orden inverso.

Para conseguir que la clase trabaje con un tipo particular de datos se define en el código la sintaxis de las líneas **124** y **126**. En ese momento el identificador *T* se particulariza para aquellos datos que se han definido en el patrón. La ventaja de las clases patrón es que se pueden generar todo tipo de clases con distintos tipos de datos, tal y como se ha hecho en el código; cualquier otra estrategia pasaba por definir dos clases distintas duplicando el código para cada tipo de datos.

Al igual que las funciones patrón, una clase patrón define un conjunto de clases limitadas que necesitan crearse para ser utilizadas como clases normales. El uso de *templates* con clases es más común que con funciones y, por tanto, se usa más a menudo, pero no demasiado porque suelen relentizar el código.

Aunque son conceptos bastante parecidos, no hay que confundir clases patrón con clases abstractas. Una clase *template* es en realidad un conjunto de clases que trata distintos tipos de datos. Una clase abstracta es una sola clase que puede tratar todo tipo de datos. Cuándo usar *templates* y cuándo utilizar una combinación de herencia y polimorfismo para simular clases abstractas depende mucho del problema. Los *templates* son rápidos y pueden ser útiles para unos pocos tipos, mientras que el polimorfismo es dinámico y, por tanto, más lento, pero se acerca más a la idea pura de programación orientada a objetos. En general, los patrones no tienen un uso muy habitual y se circunscriben a entidades como matrices, vectores, listas y funciones de ordenación principalmente.

6 Ejemplo

6.1 Versión 1. Se define la clase para matrices como un *template*, ahora es posible crear matrices de enteros y de reales sin necesidad de reescribir todo el código y cambiar sólo *int* por *double.*

```
#include <iostream.h>
#include <stdlib.h> // exit(); abs();


template <class T> class TemplateMatrix
{
public:
 T* s;
 unsigned long nr, nc, size;

 TemplateMatrix (unsigned long m=0, unsigned long n=0) // constructor
       {
       if(m<0 || n<0)
              cerr << " dimension imposible, tomo valor absoluto" << endl;
       nr=abs(m); nc=abs(n); size=nc*nr;

       if (!size) { s=NULL; return;}

       s=new T[size];
       if (!s) { cerr << "no hay memoria" << endl; exit(1);}
       for (unsigned long i=0; i<size; i++) s[i]=(T)NULL;
       }


 TemplateMatrix (const TemplateMatrix& m) // copy constructor
 : nr(m.nr), nc(m.nc), size(m.nr*m.nc)
       {
       if (!size) { s=NULL; return;}
       s=new T[size];
       if (!s) { cerr << "no hay memoria" << endl; exit(1);}
       for (unsigned long i=0; i<size; i++) s[i]=(T)m.s[i];
       }

 virtual ~TemplateMatrix() {delete [] s; s=NULL;} // destructor

 T& operator () (unsigned long m, unsigned long n)
       {
       if ((m > nr || m < 1) && (n > nc || n < 1))
       { cerr << "indices fuera de limites " <<endl; exit(1);}
        return s[nc*(m-1) + n-1];
       }

 TemplateMatrix& operator = (const TemplateMatrix& m) // asignacion
 {
       if (this == &m) return *this;
       if (size != m.size)
```

```
       {
       if (s) delete s;
              s = new T[m.size];
         if (!s) { cerr << "no hay memoria" << endl; exit(1);}
       }
       nr = m.nr;   nc = m.nc;   size = m.size;
       for (unsigned long i=0; i<size; i++) s[i]=(T)m.s[i];
       return *this;
       }

 TemplateMatrix& operator += (const TemplateMatrix& m) // operator +=
       {
       if (size == 0 && m.size != 0) return *this = m;
       if ((size == 0 && m.size == 0) || (size!=m.size))
              {  cerr  <<  "sumando  matrices  de  dimensiones  erroneas"  <<  endl;
exit(1);}
       for (unsigned long i=0; i<size; i++) s[i]+=m.s[i];
       return *this;
       }

 friend TemplateMatrix operator +
                     (const TemplateMatrix& a, const TemplateMatrix &b)
       {
       TemplateMatrix <T> sum=a;
       if ((a.size == 0 && b.size == 0) || (a.size!=b.size))
              {  cerr  <<  "sumando  matrices  de  dimensiones  erroneas"  <<  endl;
exit(1);}
       sum+=b;
       return sum;
       }


 friend ostream& operator << (ostream& os, const TemplateMatrix& m)
       {
       os << "Matrix +++ "; if (!m.size) os<<"void";
       for (unsigned long i=1; i<=m.nr; i++)
       {
        os << "\n";
        for (unsigned long j=1; j<=m.nc; j++)
               os << m.s[m.nc*(i-1) + j-1] <<" " ;
       }
       os << "\n";
       return os;
       }

};


int main ()
{
 TemplateMatrix<double> A(2,1);
 int i,j;
 for (i=1; i<=2; i++)
  for (j=1; j<=1; j++)
  {cout << "coeficiente (" <<i <<"," << j<<")="; cin >> A(i,j);}

 TemplateMatrix<int> B(2,1);
 for (i=1; i<=2; i++)
  for (j=1; j<=1; j++)
  {cout << "coeficiente (" <<i <<"," << j<<")="; cin >> B(i,j);}

 cout << A << B ;

 return 0;
}
```

6.2 Versión 2. Se utiliza la clase *template* anterior y se define una herencia para vectores como en la ficha anterior. Esto quiere decir que es posible crear matrices de enteros, reales, etc. y lo mismo de vectores sin reescribir nada de código.

```
#include <stdlib.h> // exit();
#include <iostream.h>

#define Matrix TemplateMatrix<double>
#define IntMatrix TemplateMatrix<int>

#define Vector TemplateVector<double>

template <class T> class TemplateMatrix
{
public:
 T* s;
 unsigned long nr, nc, size;

 TemplateMatrix (unsigned long m=0, unsigned long n=0) // constructor
       {
       if(m<0 || n<0)
              cerr << " dimension imposible, tomo valor absoluto" << endl;
       nr=abs(m); nc=abs(n); size=nc*nr;

       if (!size) { s=NULL; return;}

       s=new T[size];
       if (!s) { cerr << "no hay memoria" << endl; exit(1);}
       for (unsigned long i=0; i<size; i++) s[i]=(T)NULL;
       }


 TemplateMatrix (const TemplateMatrix& m) // copy constructor
 : nr(m.nr), nc(m.nc), size(m.nr*m.nc)
       {
       if (!size) { s=NULL; return;}
       s=new T[size];
       if (!s) { cerr << "no hay memoria" << endl; exit(1);}
       for (unsigned long i=0; i<size; i++) s[i]=(T)m.s[i];
       }

 virtual ~TemplateMatrix() {delete [] s; s=NULL;} // destructor

 T& operator () (unsigned long m, unsigned long n) // acceso a datos
       {
       if ((m > nr || m < 1) && (n > nc || n < 1))
       { cerr << "indices fuera de limites " <<endl; exit(1);}
        return s[nc*(m-1) + n-1];
       }

 TemplateMatrix& operator = (const TemplateMatrix& m) // asignacion
 {
       if (this == &m) return *this;
       if (size != m.size)
       {
       if (s) delete s;
              s = new T[m.size];
         if (!s) { cerr << "no hay memoria" << endl; exit(1);}
       }
       nr = m.nr;   nc = m.nc;   size = m.size;
       for (unsigned long i=0; i<size; i++) s[i]=(T)m.s[i];
       return *this;
       }
```

```
 TemplateMatrix& operator += (const TemplateMatrix& m) // operator +=
       {
       if (size == 0 && m.size != 0) return *this = m;
       if ((size == 0 && m.size == 0) || (size!=m.size))
              {  cerr  <<  "sumando  matrices  de  dimensiones  erroneas"  <<  endl;
exit(1);}
       for (unsigned long i=0; i<size; i++) s[i]+=m.s[i];
       return *this;
       }

 friend TemplateMatrix operator +
                     (const TemplateMatrix& a, const TemplateMatrix &b)
       {
       TemplateMatrix <T> sum=a;
       if ((a.size == 0 && b.size == 0) || (a.size!=b.size))
              {  cerr  <<  "sumando  matrices  de  dimensiones  erroneas"  <<  endl;
exit(1);}
       sum+=b;
       return sum;
       }


 friend ostream& operator << (ostream& os, const TemplateMatrix& m)
       {
       os << "Matrix +++ "; if (!m.size) os<<"void";
       for (unsigned long i=1; i<=m.nr; i++)
       {
        os << "\n";
        for (unsigned long j=1; j<=m.nc; j++)
              os << m.s[m.nc*(i-1) + j-1] <<" " ;
       }
       os << "\n";
       return os;
       }

};


template <class T> class TemplateVector
: public TemplateMatrix <T>
{
 public:
 TemplateVector (unsigned long l) :   TemplateMatrix <T>(l,1) {}
 TemplateVector (const TemplateMatrix<T>& m) :    TemplateMatrix <T> (m) {}

 T& operator () (unsigned long m)      // acceso a datos
{
 if (m > nr || m < 1)
       { cerr << "indices fuera de limites " <<endl; exit(1);}
   return s[m-1];
 }

 friend ostream& operator << (ostream& os, const TemplateVector& v)
 {
  os << "Vector = "; if (!v.size) os<<"void";
  for (unsigned long i=0; i<v.size; i++) os << v.s[i] <<" " ;
  os << "\n";
  return os;
 }

};

int main ()
{
```

```
 Matrix A(2,1);
 for (int i=1; i<=2; i++)
  for (int j=1; j<=1; j++)
  {cout << "coeficiente (" <<i <<"," << j<<")="; cin >> A(i,j);}
 Vector B(A);

 Vector D=A+B;
 Matrix C=A+B;
 cout << A << B << C << D;

 return 0;
}
```

Ficha 15: Excepciones

1 Objetivos generales

- Comportamiento anómalo en programas. Manejo de excepciones.

2 Código de trabajo

```
//Ficha 15
#include <stdlib.h>
#include <iostream.h>

class CriticalError
{
 public:
   CriticalError(){}
   virtual ~CriticalError(){}
   virtual void debug_print()
       { cerr <<
         "DECISION:El programa va a terminar" << endl
          << "*********************** end"<<endl;
          exit(1);}
};

class BadDataInput: public CriticalError
{
 double temperature;
 public:
   BadDataInput(double dat):CriticalError(){temperature=dat;}
   void debug_print()
     {
       cerr << "*********************** informe"<<endl
   <<"PROBLEMAA: " << temperature
   <<" es un valor imposible de la temperatura en Kelvin" << endl
           <<"SUGERENCIA (1): vuelva a leer el dato"<<endl;
     CriticalError::debug_print();
     }
};


int main ()
{
   double TKelvin,TDegrees;

   cout << "Valor de la temperatura "; cin >> TKelvin;
```

```

   try // vamos a hacer algo
   {
    if (TKelvin<0.) throw BadDataInput(TKelvin);
    TDegrees=TKelvin-273.;
   }

   catch(CriticalError& Err) // tratamos el error
   {
    Err.debug_print();
   }

   cout << "La temperatura en centigrados es " << TDegrees <<endl;
   return 1;
}
```

3 Conceptos

3.1 Comportamiento anómalo del programa

En general, todo programa suele hacer algo y el programador es responsable directo de que así sea, a través de la creación, compilación y depuración del código. No obstante, a veces, los programas se comportan de forma errónea bajo ciertas circunstancias. Por ejemplo, si el programa pide la introducción de un dato positivo y el usuario escribe un valor negativo, como puede darse en el caso del ejemplo superior, este hecho tan sencillo no invalida el funcionamiento del programa, pero sí la validez de sus resultados. En consecuencia, dejar que el programa se ejecute a pesar de que los resultados van a ser erróneos no tiene ningún sentido, por lo tanto es tarea del programador prever en qué ocasiones se puede producir un hecho anómalo. En el sencillo ejemplo del código, si el valor de la temperatura en grados Kelvin toma un valor negativo, el programa realizará un cálculo erróneo, por ello se debe tomar alguna decisión.

El comportamiento anómalo se identifica con errores en tiempo de ejecución. Cada programa es diferente y tiene sus problemas particulares, pero, en general, aquellas ocasiones en que un programa puede conducir a un comportamiento anómalo y es conveniente tener en cuenta y acotar son:

Errores en la introducción de datos
Este tipo de error es muy común entre los usuarios de un programa, sobre todo si los datos se leen de un fichero con un formato específico. Puede suceder en un porcentaje muy alto de casos que no existan suficientes campos con datos, o bien que el orden no sea el correcto, incluso que las magnitudes sean incorrectas; en cualquiera de estas situaciones el programa debe ser capaz de detenerse e informar al usuario del error en la lectura de los datos. En ocasiones y para cierto tipos de programas, es recomendable establecer un valor por defecto de todas las variables para evitar que el programa se detenga, aunque siempre se debe avisar de que algunos valores se han inicializado por defecto.

Errores de división por cero
Lógicamente es una situación que hará estallar el programa deteniendo su ejecución. Para evitar este inconveniente se deberá siempre comprobar que el divisor es distinto de cero antes de realizar la operación. Si este fuera el caso, el programador puede avisar del error, evitar la operación y continuar el programa, o bien detenerlo si fuera imprescindible dicha operación. En este último caso se debe avisar al usuario de los motivos por los cuales se produce el error, tal vez una introducción de datos incorrecta, etc.

Errores con punteros
Este tipo de errores es muy problemático y difícil de detectar. Por ello la programación de punteros es un arma de doble filo; mejoran la velocidad, pero si están mal programados el error cuesta mucho de detectar. Posibles fuentes de error con punteros son: mala asignación, no inicialización y sin reserva de memoria dinámica. Es importante gestionar bien la memoria dinámica y evitar punteros innecesarios incluyendo funciones ya definidas en la librería.

Errores de salida en un bucle
Este error es responsabilidad directa del programador, ocurre cuando el incremento del contador está mal gestionado, o bien cuando existe una condición lógica imposible. En este caso el programa suele quedarse 'colgado' ejecutando la misma sentencia. No son errores peligrosos porque el compilador o el sistema siempre permite cortar los bucles infinitos mediante una tecla especial, sin embargo son de difícil gestión por parte del mismo programa.

Errores en condicionales
Este error también depende de una mala programación, en general alguna condición lógica no contemplada. Por ello siempre es conveniente utilizar un *else* para hacer algo en cualquier otro caso. Es importante que los *if*, *switch* o *while* condicionales cierren todas las puertas.

Desbordamiento de pila
En este caso el sistema suele abortar el programa y detener su ejecución; desde el propio programa suele ser difícil evitar este error porque la pila se gestiona desde el sistema operativo. Para evitarlo se recomienda no tener muchos archivos abiertos ni variables globales.

Errores de límites
Esta circunstancia está muy relacionada con los valores de los datos; bien puede ser que la introducción e inicialización de valores sea correcta o puede ser que en tiempo de ejecución alguna variable cambie su valor entrando en un dominio de definición incorrecto. Para contemplar el error basta con verificar la magnitud de la variable después de que una operación modifique su valor.

Falta de memoria (interna o externa)
Si el programa requiere mucha memoria, aunque se gestione de forma dinámica, puede que el ordenador no sea capaz de suministrarla, ni siquiera a través de disco. Entonces el sistema operativo detendrá el programa porque no puede satisfacer los requerimientos. Para evitar este inconveniente se debe colocar alguna instrucción que ejecute un código alternativo en caso de no poder dimensionar la memoria solicitada. En general, los errores de memoria están asociados a la creación/destrucción de objetos temporales, por lo tanto este tipo de error es común en el código de constructores, destructores, operadores de asignación, o constructores copia. Siempre se debe verificar la correcta construcción/destrucción de objetos.

Conversiones
A veces se realizan comparaciones lógicas de igualdad entre tipos diferentes, por ejemplo *int* y *double*. También se puede ejecutar una función pasando un argumento que no es del tipo prototipado. En general estos problemas se pueden soslayar mediante *castings* o definiendo conversiones entre los tipos creados por el usuario. En cualquier caso se debe tener constancia de que una conversión se realiza y de verificar su correcta ejecución.

Errores de portabilidad
Este error suele ser difícil de tratar en tiempo de ejecución, pero se puede obviar en tiempo de compilación. Para ello basta con utilizar las predirectivas de compilación y crear versiones diferentes en función de la plataforma de trabajo.

3.2 Tratamiento común de errores

El buen diseño de un programa pasa por detectar y evitar el funcionamiento anómalo. En general, un buen programador tiene en cuenta las posibles aberraciones que pueden darse: no inicialización de variables o punteros, datos erróneos, etc. y da un tratamiento a cada uno de ellos. Una buena estrategia pasa por programar cada clase o función de manera que detecte sus errores, haga algo y retorne un código con información del estado en el que se ha acabado una determinada operación. En función del error será conveniente detener el programa o, por el contrario, continuar con su ejecución; además será conveniente imprimir información acerca del error utilizando el *stream cerr*. En el ejemplo se muestra una posible impresión de comentarios en las líneas **010** y **022**.

El retorno de un código de error significa que se tendrá que clasificar y tratar a través de alguna estrategia tipo *switch* y, dado que los programas suelen ser largos, normalmente habrá muchos códigos de error diferentes, por lo que la complejidad del tratamiento puede ser grande. Atención también, porque si el error es muy crítico y no se puede continuar, la mejor opción pasa por salir y detener el programa. Sin embargo, no es aconsejable utilizar alguna de las funciones de salida súbita como *exit* (como en la línea **013**) que están en la cabecera *stdlib.h* **001**, ya que al salir del programa se puede dejar memoria sin liberar, ficheros sin cerrar, objetos sin destruir, etc. Para evitar esto último hay que decir que existe la función *atexit*, que permite registrar funciones que se deban llamar antes de la finalización del programa. En cualquier caso, se observa que las estrategias pasan por tener listas de códigos de error y salidas difíciles de controlar.

3.3 Excepciones

En C++ se define e*xcepción* como un mensaje de anomalías en tiempo de ejecución, es decir, algo para avisar al propio programa de una posible fuente de error como la división por cero, el direccionamiento incorrecto, la falta de memoria, etc. Un programa fiable debe controlar estas excepciones y actuar en consecuencia para evitar errores en los resultados. El C++ (a partir de la versión 3.0) dispone de un *manejador de excepciones.*

En el ejemplo se tratan las excepciones desde clases; por supuesto esto no tiene porque ser así, pero se entiende que enlaza mejor con la filosofía orientada a objetos del C++. Como se ha dicho anteriormente, el posible error en el programa se produce cuando el dato de entrada, la temperatura en grados Kelvin, es menor que cero. Por lo tanto se van a definir unas clases que harán algo con respecto al error. En concreto, en la línea **004** se define una clase base *CriticalError* que se encarga de abortar el programa mediante la instrucción *exit* en la línea **013**. Por otra parte, en la línea **016** una clase derivada *BadDataInput* representa al error tipo, en este caso una introducción errónea de datos. La detección del error se realiza a través de un *if* lógico situado en la línea **039**.

Nótese que este tipo de error no tiene porqué finalizar con una salida del programa, bastaría con tomar el valor absoluto de la variable *TKelvin*, realizar el cálculo y que la excepción sólo diera un aviso sin llegar a abortar la ejecución. Por supuesto, el criterio de programación es personal y esta segunda opción sería también totalmente válida.

El concepto de excepción, ligado al del comportamiento anómalo o no deseable del programa, tiene una clara metodología de tratamiento en C++. Básicamente, hay que seguir el siguiente procedimiento; en primer lugar detectar el posible error y, una vez que se produce, lanzar la excepción. En segundo lugar, una vez se ha lanzado, se debe recoger en alguna parte del programa y darle un tratamiento. La idea es clara, lanzamiento de la excepción (detección del error) y tratamiento (qué debe hacer el

programa); para trabajar con excepciones se definen tres nuevas palabras clave en C++: *try, catch* y *throw.*

El bloque de prueba: try
La definición de un bloque *try* sirve para delimitar la zona de código donde se puede producir una excepción. El bloque *try* va a intentar algo; si sale bien el código continúa, si no se lanza una excepción. En la línea **037** se inicia aquello que se quiere hacer, en particular calcular la temperatura en grados centígrados, esa operación se realiza en la línea **040**. Sin embargo, es deseable que se realice si, y únicamente si, el dato es positivo o nulo, en caso contrario se producirá un error.

El lanzamiento de la excepción: throw
Si se detecta un error dentro del bloque *try* se puede lanzar la excepción mediante la sentencia *throw*. En ese momento la excepción sale al terreno de juego esperando que alguien la recoja y haga algo con ella. En la línea **037** se verifica el valor en grados Kelvin; si el dato es erróneo, se lanza la excepción en la misma línea. Nótese que el error es del tipo *BadDataInput*.

La recogida: catch
La sentencia *catch* permite recoger la excepción y darle un tramiento. Si el error es muy grande, el bloque *catch* cerrará todos los ficheros, liberará memoria, etc. antes de terminar el programa y escribir el último mensaje de error. Si el error es pequeño, únicamente escribirá un aviso y continuará con el proceso. A veces, dentro del bloque no se puede resolver el problema, entonces se deberán realizar otros lanzamientos o dejar que otro bloque capture la excepción. De hecho, cuando definimos un *catch* estamos indicando que en ese nivel se recogerá cualquier error que se haya producido por debajo de él.

En el ejemplo, la recogida se realiza en la línea **043** y simplemente llama a una función de la clase. Nótese que se captura a través de la base, el argumento es del tipo *CriticalError*, pero por el mecanismo de la herencia se imprime el mensaje de la derivada, función *debug_print* en línea **020**. Además, al final la derivada también llama a la base en línea **026** y termina el programa en **013**. En lugar de terminar en **049**, que sería lo deseable.

Se observa que, dependiendo del tipo de *throw*, se entra en el *catch* apropiado; si hubieran diferentes tipos de error se realizarían diferentes capturas. Para ello, se siguen las mismas normas que las definidas para la sobrecarga de funciones y, por lo tanto, se empieza a comparar en el orden en el que están puestos los *catch*. Cuando uno coincide con el tipo de *throw*, se ejecuta y se obvian todos los demás hasta el siguiente bloque ejecutable que no sea un *catch*.

Hay que destacar que no puede haber más de un error en curso en un momento dado, ya que cuando se hace un lanzamiento se va subiendo de nivel hasta que se trata y en último caso hasta salir del *main ( ).*

4 Ejercicios

4.1 Definir excepciones para la clases de polígonos.

Solución al ejercicio

```
/*control de una operacion imposible*/

//Ficha 15
#include <iostream.h>
#include <string.h> // para la funcion strcp ()

class CriticalError
{
```

```
public:
  int ErrLine; char Fig[50];

  CriticalError(int line,char *Buffer){ErrLine=line; strcpy(Fig,Buffer);}
  void PrintIt(void)
  {cerr << "Error en la linea " << ErrLine
        <<" operacion imposible para un " << Fig << endl;}
  ~CriticalError () {}
};


class abstracta
{
 public:
      abstracta (void){}
      virtual double area () = 0;
      virtual double volumen () =0;
};


class poliedro : public abstracta
{
public:
      poliedro () : abstracta () {};
      ~poliedro () {}
      double area () { CriticalError Bye(__LINE__,"poliedro"); throw Bye; return
0.;}
      double volumen () {return 0.;}
};

class cubo : public poliedro
{
public:
 double c;
 cubo (double lado) : poliedro () {c=lado;}
 ~cubo () {}
 double volumen(void) {cout << "volumen del cubo ="; return c*c*c;}
};


class poligono : public abstracta
{
 public:
  poligono (void) : abstracta () {}
 ~poligono () {}
 double area () {return 0.;}
 double volumen () {CriticalError Bye(__LINE__,"poligono");
                   throw Bye;     return 0.;}
};

class rectangulo : public poligono
{
public:
      double a,b;
      rectangulo (double lado1, double lado2): poligono ()
            { a=lado1; b=lado2;}
      virtual ~rectangulo () {}
      double area(void) {cout << "area del rectangulo ="; return a*b;}
};


class cuadrado : public rectangulo
{
public:
      cuadrado (double lado) : rectangulo (lado,lado) {}
```

```
       ~cuadrado () {}
       double area(void) {cout << "area del cuadrado ="; return a*a;}
};


int main (void)
{
 rectangulo r1 (1.,1.); cuadrado c1 (2.);
 cubo q1(2.);

 try
 {
  r1.volumen();
 }
 catch (CriticalError& er)
 {
  er.PrintIt();
 }

 try
 {
  q1.area();
 }
 catch (CriticalError& er)
 {
  er.PrintIt();
 }

 return 0;
}
```

5 Ejemplo

Se definen las clases de la ficha anterior con el tratamiento de errores mediante excepciones.

```
//Ficha 15
#include <iostream.h>
#include <stdlib.h> // exit() abs();

// definiciones para simplificar el codigo main
#define Matrix TemplateMatrix<double>
#define ReaMatrix TemplateMatrix<double>
#define IntMatrix TemplateMatrix<int>

#define Vector TemplateVector<double>


// * * * * * * * * * * * INICIO EXCEPCIONES * * * * * * * * * * * * * * *

//---------------------------------------------------- INICIO EXCEPCIONES
// definiendo las posibles excepciones del programa
// error critico general:                    CriticalError
//     error de desborde de memoria:         OverFlow
//     error de fuera de limites:            OutOfRange
//     error de dimensiones incompatibles:   AddBadDim

//--------------------------------- INICIO ERROR CRITICO
class CriticalError
{
public:
   CriticalError(){}
   virtual ~CriticalError(){}
   virtual void debug_print()
     {
```

```
      cerr << "DECISION: Es un error critico no asumible."
           <<" Lo siento pero el programa va a terminar. "<<endl
           << "************************ fin"<<endl;
        exit(1);
      }
};
//---------------------------------- FINAL ERROR CRITICO

//---------------------------------- INICIO ERROR OVERFLOW
class OverFlow: public CriticalError
{
   int size;
public:
   OverFlow(int l):CriticalError(){size=l;}
   void debug_print()
    {
     cerr << "************************ informe"<<endl
          <<"PROBLEMA: no hay memoria para una matriz de "
          <<size<<endl
          <<"SUGERENCIA (1): cierre otras aplicaciones"<<endl
          <<"SUGERENCIA (2): intente un problema mas pequeño"<<endl
          <<"SUGERENCIA (3): cambie la configuracion de hardware"<<endl;
     CriticalError::debug_print();
    }
};
//---------------------------------- FINAL ERROR OVERFLOW

//---------------------------------- INICIO ERROR OUTOFRANGE
class OutOfRange: public CriticalError
{
   int i,j;
public:
   OutOfRange(int m, int n):CriticalError(){i=m; j=n;}
   void debug_print()
    {
     cerr << "************************ informe"<<endl
          <<"PROBLEMA: accediendo a un indice fuera de la matriz "
          <<"("<<i<<","<<j<<")"<<endl
      <<"SUGERENCIA (1): cuidado con las dimensiones de la matriz"
      <<endl
      <<"HIPOTESIS: puede crear conflictos con otras partes del programa"
      <<endl;
     CriticalError::debug_print();
    }
};
//---------------------------------- FINAL ERROR OUTOFRANGE

//---------------------------------- INICIO ERROR ADDBADDIM
class AddBadDim: public CriticalError
{
 int nr1,nr2;
 int nc1,nc2;
public:
   AddBadDim(int nrows1,int ncol1,int nrows2, int ncol2)
       :CriticalError(){nr1=nrows1; nc1=ncol1; nr2=nrows2; nc2=ncol2;}
   void debug_print()
    {
     cerr << "************************ informe"<<endl
          <<"PROBLEMA: sumando matrices de diferente tamaño A("
          <<nr1<<","<<nc1<<")+B("<<nr2<<","<<nc2<<")"<<endl
          <<"SUGERENCIA (1): cuidado con las dimensiones de la matriz "
           <<endl
          <<"HIPOTESIS: operacion imposible"<<endl;
     CriticalError::debug_print();
    }
```

```
};
//--------------------------------- FINAL ERROR ADDBADDIM

//---------------------------------------------------- FINAL EXCEPCIONES
// * * * * * * * * * * * * FINAL EXCEPCIONES * * * * * * * * * * * * * *

// * * * * * * * * * * * INICIO TEMPLATE MATRIX * * * * * * * * * * * * *
//----------------------------------------------------- INICIO TEMPLATES

template <class T> class TemplateMatrix
{
public:
//--------------------------------- INICIO DATOS TEMPLATE MATRIX
 T* s;
 int nr, nc, size;
//--------------------------------- FINAL DATOS TEMPLATE MATRIX

//--------------------------------- INICIO CONSTRUCTOR TEMPLATE MATRIX
 TemplateMatrix (int m=0, int n=0) // constructor
 {
   if(m<0 || n<0)
     cerr << "*********************** informe"<<endl
     <<"PROBLEMA: creando una matriz de dimensiones negativas filas ="
     <<m<<" columnas="<<n<<endl
     <<"HIPOTESIS: asumiendo un error ignorable en linea ="
     <<__LINE__<<" en archivo ="<<__FILE__<<endl
     <<"DECISION: tomando valores absolutos"<<endl
     << "*********************** end"<<endl;
  nr=abs(m); nc=abs(n); size=nr*nc; // valores iniciales
  if (!size) { s=NULL; return;}
  try
    {
     s=new T[size];
     if (!s) {cerr << "*********************** informe"<<endl
             <<"OJO en linea ="<<__LINE__<<" en archivo "<<__FILE__<<endl;
              throw OverFlow(size);}
     for (int i=0; i<size; i++) s[i]=(T)NULL;
    }
   catch(CriticalError& Err)
    {
     Err.debug_print();
    }
 }
//--------------------------------- FINAL CONSTRUCTOR TEMPLATE MATRIX


//------------------------------ INICIO CONSTRUCTOR COPIA TEMPLATE MATRIX
TemplateMatrix (const TemplateMatrix& m) // copy constructor
: nr(m.nr), nc(m.nc), size(m.nr*m.nc)
{
 if (!size) { s=NULL; return;}
 try
    {
   s=new T[size];
   if (!s) {cerr << "*********************** informe"<<endl
            <<"OJO en linea="<<__LINE__<<" en archivo="<<__FILE__<<endl;
            throw OverFlow(size);}
   for (int i=0; i<size; i++) s[i]=(T)m.s[i];
   }
 catch(CriticalError& Err)
   {
    Err.debug_print();
   }
}
//------------------------------ FINAL CONSTRUCTOR COPIA TEMPLATE MATRIX
```

```
//------------------------ INICIO CONSTRUCTOR ASIGNACION TEMPLATE MATRIX
TemplateMatrix& operator = (const TemplateMatrix& m)
{
 if (this == &m) return *this; // si el el mismo vuelve
 if (size != m.size)
   {
   try
      {
       if (s) delete s;
       s = new T[m.size];
       if (!s) {cerr << "*********************** informe"<<endl
           <<"OJO en linea="<<__LINE__<<" en archivo="<<__FILE__<<endl;
            throw OverFlow(m.size);}
      }
   catch(CriticalError& Err)
       {
        Err.debug_print();
       }
   }
nr = m.nr;
nc = m.nc;
size = m.size;
for (int i=0; i<size; i++) s[i]=(T)m.s[i];
return *this;
}
//------------------------ FINAL CONSTRUCTOR ASIGNACION TEMPLATE MATRIX

//------------------------------ INICIO DESTRUCTOR TEMPLATE MATRIX
virtual ~TemplateMatrix() {delete [] s; s=NULL;} // destructor
//------------------------------ FINAL DESTRUCTOR TEMPLATE MATRIX

//-------------------------------------- INICIO OPERADORES TEMPLATE MATRIX
//------------------------------ INICIO ACCESO A DATOS
T& operator () (int m, int n) // acceso con estilo Fortran
{
 try
    {
    if (m > nr || n > nc || m < 1 || n < 1)
           {cerr << "*********************** informe"<<endl
            <<"OJO en linea="<<__LINE__<<" en archivo="<<__FILE__<<endl;
            throw OutOfRange(m,n);}
    return s[nc*(m-1) + n-1];
   }
 catch(CriticalError& Err)
   {
    Err.debug_print();
   }
}
//------------------------------ FINAL ACCESO A DATOS


//--------------------------------- OPERADOR SUMA DE MATRICES
// operator A += B
TemplateMatrix& operator += (const TemplateMatrix& m)
{
 try
   {
   if (size == 0 && m.size == 0)
       {cerr << "*********************** informe"<<endl
        <<"PROBLEMA: intentando sumar matrices nulas en linea="
        <<__LINE__<<" en archivo="<<__FILE__<<endl
       <<"SUGERENCIA (1): cuidado con las dimensiones de la matriz "<<endl
       <<"DECISION: Continua la operacion, atencion a los resultados"<<endl
       << "*********************** end"<<endl;}
```

```
   if (size!=m.size)
        {cerr << "*********************** informe"<<endl
        <<"OJO en linea="<<__LINE__<<" en archivo="<<__FILE__<<endl;
             throw AddBadDim(nr,nc,m.nr,m.nc);}
   for (int i=0; i<size; i++) s[i]+=m.s[i];
    }
catch(CriticalError& Err)
     {
    Err.debug_print();
   }
 return *this;
} // fin de +=

// operator A + B
friend TemplateMatrix operator +
                     (const TemplateMatrix& a, const TemplateMatrix &b)
{
TemplateMatrix <T> sum=a;
try
   {
   if (a.size == 0 && b.size == 0)
              {cerr << "*********************** informe"<<endl
         <<"PROBLEMA: intentando sumar matrices nulas en linea="
         <<__LINE__<<" en archivo="<<__FILE__<<endl
        <<"SUGERENCIA (1): cuidado con las dimensiones de la matriz "<<endl
        <<"DECISION: Continua la operacion, atencion a los resultados"<<endl
        << "*********************** end"<<endl;}
    if (a.size!=b.size)
        {cerr << "*********************** informe"<<endl
        <<"OJO en linea="<<__LINE__<<" en archivo="<<__FILE__<<endl;
             throw AddBadDim(a.nr,a.nc,b.nr,b.nc);}
   sum+=b;
   }
catch(CriticalError& Err)
     {
    Err.debug_print();
   }
return sum;
} // fin de +
//----------------------------------- FINAL OPERADORES SUMA DE MATRICES

//----------------------------------- INICIO SALIDA DATOS
friend ostream& operator << (ostream& os, const TemplateMatrix& m)
{
 os << "Matrix +++ "; if (!m.size) os<<"void";
 for (int i=1; i<=m.nr; i++)
   {
    os << "\n";
    for (int j=1; j<=m.nc; j++)
        os << m.s[m.nc*(i-1) + j-1] <<" " ;
   }
 os << "\n";
 return os;
 }
//----------------------------------- FINAL SALIDA DATOS

}; // fin de la clase
// * * * * * * * * * * * FINAL TEMPLATE MATRIX * * * * * * * * * * * * *

// * * * * * * * * * * * INICIO TEMPLATE VECTOR * * * * * * * * * * * * *

template <class T> class TemplateVector
: public TemplateMatrix <T>
{
 public:
```

```
 TemplateVector (int l) :  TemplateMatrix <T>(l,1) {}

    TemplateVector (const TemplateMatrix<T>& m) :
       TemplateMatrix <T> (m) {}

//------------------------------ INICIO ACCESO A DATOS
T& operator () (int m)
{
 if (m > nr || m < 1)
  {cerr << "************************ informe"<<endl
        <<"OJO en linea="<<__LINE__<<" en archivo="<<__FILE__<<endl;
        throw OutOfRange(m,1);}
   return s[m-1];
 }
//------------------------------ FINAL ACCESO A DATOS
//--------------------------------- INICIO SALIDA DATOS
friend ostream& operator <<
(ostream& os, const TemplateVector& v)
 {
  os << "Vector = "; if (!v.size) os<<"void";
  for (int i=0; i<v.size; i++) os << v.s[i] <<" " ;
  os << "\n";
  return os;
 }
//--------------------------------- FINAL SALIDA DATOS
}; // final clase vector
// * * * * * * * * * * * FINAL TEMPLATE VECTOR * * * * * * * * * * * * *
//------------------------------------------------------- FINAL TEMPLATES
//--------------------------------------------------- INICIO DEL PROGRAMA
int main ()
{
 int i,j;
 Matrix A(2,2);
 for (i=1; i<=2; i++)
  for (j=1; j<=2; j++)
  {cout << "coeficiente (" <<i <<"," << j<<")="; cin >> A(i,j);}
 Matrix B(2,1);
 for (i=1; i<=2; i++)
  for (j=1; j<=1; j++)
  {cout << "coeficiente (" <<i <<"," << j<<")="; cin >> B(i,j);}

 cout << A << B;

 Matrix C=A+B; // lanza la excepcion

 return 0;
}
//------------------------------------------------------- FIN DEL PROGRAMA
```

6 Ampliación de conceptos

6. 1 Otros lanzamientos

En el ejemplo del código de trabajo se ha asociado el manejo de la excepción a una clase. En general esto no tiene por qué ser así, se puede lanzar una excepción como un tipo de variable por defecto en el lenguaje, un *int*, o bien un tipo derivado, como un *string*. Lógicamente, la sentencia *catch* tampoco tiene que recibir como argumento una clase de forma obligatoria, sin embargo el hecho de utilizar clases para organizar el tratamiento de errores supone una mejora clara. ¿Qué sentido tiene hacer un código orientado al objeto intentando mejorar la programación y dejar que el manejo de errores sea confuso y complicado? En este sentido nos parecía un contrasentido enfocar el tratamiento de excepciones desde fuera del concepto orientación a objeto.